Towards Rule Interchange and Rule Verification

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik

der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Magister der Mathematik in der Fachrichtung Angewandte

Mathematik und Informatik

Sergey Lukichev

Geboren am 15. Februar 1979 in Temirtau (Kazachstan)

Gutachter: Prof. Gerd Wagner

Gutachter: Prof. Pascal Hitzler

Gutachter: Prof. Grzegorz Nalepa

Tag der mündlichen Prüfung: 11. Februar 2010

Abstract

Rules are a critical technology component for the early adoption and applications of knowledge-based techniques in e-business, especially enterprize integration and B2B e-commerce. They also play an important role in information systems engineering, especially in the specification of functional requirements where business rules are the foundation for capturing and modeling business application logic.

When using rules, companies may encounter obstacles with two issues: The problem of rule interoperability, which is caused by a variety of rule languages and rule systems, and the problem of rule quality as a consequence of a large amount of business rules created and used in an organization.

A particular solution to the rule interoperability problem is a standardized way of performing rule interchange between different rule languages and tools. The thesis addresses the problem by considering a rule interchange mapping from the Object Constraint Language (OCL) into the Semantic Web Rule Language (SWRL). This mapping is useful for various communities. For instance, software developers, who actively use UML/OCL, may employ the mapping in order to translate their rules to SWRL and use them in a Semantic Web application. On the other hand, the research on rule interchange is interesting for Semantic Web practitioners, who work in the area of formal semantics of rule languages and have interest in the rule interchange standardization. The main contribution of the thesis concerning rule interchange is the proof of correctness of the mapping from the semantical point of view. The problem of semantic correctness of rule interchange mapping is formulated for two rule languages and solved for OCL and SWRL. The approach can be applied to other rule languages with formal semantics.

The quality of rules is high if they are expressed in the right way and express what business people want to express. However, due to various reasons, for instance communication problems between business people and rule modelers, rules may become inconsistent, incomplete or redundant. Therefore, organizations need rule quality measurement and technologies to improve the quality. A particular way to control and to improve the rule quality is by means of rule verification. In this respect, the main contribution of the thesis is the declarative rule verification approach, which can be used for detection of different problems in rule bases. The verification approach is implemented for Jena rules, which makes it more applicable for the quality control of upcoming Semantic Web rule-based applications.

Acknowledgements

I would like to use this opportunity to express my sincere acknowledgments to the people who provided support to my doctoral work in the past five years.

First of all, I would especially thank my advisor Prof. Gerd Wagner for giving me a chance to do the research at Brandenburg University of Technology Cottbus. Without his great patience, guidance, and valuable support, this thesis would have been impossible to complete.

Furthermore, my special thanks go to the members of the REWERSE Working Group I1 for their fruitful discussions and numerous supports. I thank Adrian Giurca, Oana Nicolae, and Mircea Diaconescu.

My thanks also extend to Dr. Pascal Hitzler and Dr. Grzegorz J. Nalepa for their useful advices and supportive feedbacks.

Finally I would like to thank my parents and the sister in Kazakhstan for their continual encouragement and support.

Contents

Chapter 1

Introduction

1.1 Background

According to [80], a business rule *is a directive intended to influence or guide business behavior*. In other words, business rules are statements that describe whether something can or cannot be done or give criteria and conditions for making a decision. The SBVR standard [69] defines a **business rule** as *a rule that is under business jurisdiction*. Business rules are usually expressed in a natural human language (for instance, English) or visually. A rule-based software system is used to process rules automatically and let them guide business behavior. The rules in a software system are represented formally using some rule representation language. A process of rules transformation from (semi)formal representation to formal representation is called *rule capturing*. Particular types of formally represented rules are *production rules* ([6], [88], [91]), in the form of **if** <condition> **then** <action>, and *integrity rules*, also called *integrity constraints*.

Examples of production rule systems are Jena [3], JBoss Rules [1], Oracle Business Rules [70], and ILOG JRules [9]. The Object Constraint Language (OCL) by OMG is an example of a language for expressing integrity rules [4]. In the scope of this research, related to Semantic Web technologies, there are various knowledge and rule representation languages. In particular, the Semantic Web Rule Language (SWRL) [8] and Web Ontology Language (OWL) [5] can be used for expressing integrity constraints.

Production rules can be used to represent a large variety of business rules and are widely used in automation of such business processes as mortgage, insurance, rental and other services. They are often used in conjunction with integrity rules in one knowledge base: Depending on different conditions, production rules execute actions, while integrity rules control the data consistency. Having a rule-based system with a knowledge base which includes both production rules and integrity rules, the business may encounter a number of rule quality problems such as rule consistency, completeness and inefficiency. In order to detect these problems, rules should be verified. The necessity for high-quality rules, which can be achieved with the help of verification, is debated in [85].

Another arising issue among rule application developers and rule system vendors is a problem of rule interoperability. In many cases, business rules, initially formalized in one rule language, have to be translated to another rule language. Such translation should be performed according to a standard, the purpose of which is "to allow rules to be translated

between rule languages and thus transferred between rule systems" [78].

1.1.1 Verification of Production Rules

The advantage of rule-based systems is flexibility and easiness of maintenance. The core of such systems is a knowledge base, which is, informally, a set of representations of facts about the world and may include business rules, a business vocabulary and facts. Rules can easily be added, removed or modified in the knowledge base, which allows a quick response to new business challenges and reduces expenses, needed, for instance, for compliance with new regulations or new marketing strategies. However, building a knowledge base is an incremental process where expertise is transferred from the business expert through the rule or knowledge engineer into the computer system. The initial knowledge acquisition for the knowledge base and capturing of business rules is performed by business experts, who normally keep only business goals in mind and are not usually familiar with logical formalisms and recommended rule design principles. Therefore, problems may arise during the knowledge formalization process. For instance, the knowledge may be incomplete, inconsistent or contain errors. Even if the knowledge provided by the business expert is correct and complete, it may not be correctly formalized for the further processing by the rule-based system because of various communication problems between the expert and the knowledge engineer. In order to prevent errors while executing rules, rules should be verified.

> *Verification* is a process that aims for the detection of anomalies in a rule base without considering the 'meaning' of the rules.

Domain independent verification approaches are typically based upon the concept of an *anomaly*, which is an unusual use of the knowledge representation scheme. An anomaly is not necessarily an error, but rather a potential error - it may be an actual error that needs correcting or may alternatively be intended. Some anomalies simply cause efficiency or maintenance problems, while others lead to unexpected or incorrect results.

In this work, we present a declarative (rule-based) approach to rule verification (Chapter 4). Anomalies are detected by means of so called verifier rules, which, in contrast to business rules, do not solve problems in a business domain, but analyze business rules for possible errors.

1.1.2 Rule Interchange

Due to the variety of rule languages and rule systems, rule application developers run afoul of the general problem of rules interoperability, in particular with the rule interchange problem. We can identify two reasons for the demand of rule interchange methodologies and standards:

- Many existing rule systems do not have formal semantics for rules (for instance, production rule systems), which causes problems when a company wants to switch from one rule platform to another. Since the formal meaning of rules on the source rule platform is not clear, it is difficult to make sure that the same rules on the

target rule platform will be executed as expected. This problem can be addressed by providing a standard, which fixes semantics and guarantees the expected rule execution effect.

- Even if the semantics of rule languages are defined formally (for instance, UML/OCL, OWL/SWRL, F-Logic, etc), there can be a need for translating rules from one rule language to another. In order to satisfy this need, a rule interchange methodology, based on a rule interchange standard, has to be employed.

- Due to active development of new knowledge representation languages for the Semantics Web, like RDF, OWL, and SWRL, there is a need for employing existing tools and approaches in rules engineering for the Semantic Web. A good example here is the UML/OCL modeling approach, which is actively used by the software development community, while there is a lack of tools and methodologies for modeling Semantic Web ontologies and rules. For instance, the rule interchange between UML/OCL and OWL/SWRL may help UML modelers to start with the knowledge engineering for the Semantic Web by employing existing UML technologies and tools.

In this work we define a rule interchange mapping from the Object Constraint Language (OCL) into the Semantic Web Rule Language (SWRL) and prove that this mapping is correct with respect to formal semantics of these languages (Chapter 3).

1.1.3 General Purpose Rule Markup Language

The basis for the efficient, loss-free interchange is the rule language, which satisfies several requirements. The W3C requirements to the Rule Interchange Format are stated in the"RIF Use Cases and Requirements" [78]. The following requirements are considered to be important:

- The rule language must have an XML concrete syntax. This requirement is important for the implementability by well understood techniques like XSLT and XQuery. Modern XML technologies are advanced and have variety of parsers and parser generators, which makes the usage of the language efficient and flexible.

- The rule language must cover the set of existing rule languages and different rule types like derivation rules, production rules, integrity rules and reaction rules.

- In order to provide interchange between relational languages like SWRL and functional languages like OCL, using relatively simple translators, the rule interchange language must integrate properties of the both, functional and relational languages.

- The rule language must allow different semantics for rules. Since existing rule languages have different formal semantics, the interchange language should be able to preserve any. The RIF by W3C has a default semantics in Horn Logic. However, this may not be sufficient in some cases, therefore allowing different semantics for rules is important.

Another need for the general rule markup language is imposed by the Semantic Web with its stack of technologies like OWL and RDF. The main principles for the web rule language, called"golden rules" are formulated in [90]. There are a number of rule languages, designed for the Semantic Web, for instance, RuleML and OWL/SWRL. These languages accommodate web concepts like URIs and XML namespaces. Therefore, the general purpose rule language is also required to accommodate:

- Web naming concepts, such as URIs and XML namespaces;
- The ontological distinction between objects and data values;
- The datatype concepts of RDF and user-defined datatypes.

The language, which satisfies the above requirements is presented in Chapter 2 and called REWERSE I1 Rule Markup Language (R2ML). We use MOF/UML to model the language and explain how R2ML satisfies the requirements. We employ R2ML in Chapter 3 for defining interchange mappings from OCL to SWRL.

1.2 Research Objectives

Current research has two main objectives:

1. To provide a mapping from one rule language to another via an interchange format and prove that the mapping is correct with respect to formal semantics of these languages.
2. To provide a declarative approach for the verification of production rules.

In order to reach the first objective we will consider particular rule languages: OCL and SWRL. We will analyze the syntax and semantics of these languages, build interchange mappings and explain how the correctness of these mappings can be provided. We will define the concept of the semantic correctness of a rule interchange mapping in Section 3.2.

For the second objective, we will define a number of anomalies which have been discovered heuristically and are known among rule modelers. These anomalies are detected by means of verifier rules. The execution result of these rules is a set of detected anomalies.

1.3 The Structure of the Thesis

The thesis contains three main chapters: the definition of the syntax and semantics of the general-purpose rule markup language R2ML (Chapter 2), the rule interchange between OCL and SWRL (Chapter 3) and verification of production rules (Chapter 4).

1.3.1 Chapter 2: REWERSE Rule Markup Language

This chapter describes the rule markup language R2ML, which is used as an intermediate representation of rules in the interchange between OCL and SWRL. Section 2.1 provides an

overview of related works on rule interchange, other rule languages, and gives a motivation for our choice of R2ML as an interchange language for rules. Section 2.1.2 describes main principles used for designing R2ML and explains the modeling approach by means of MOF/UML. Section 2.2 presents the language by defining rule constructs bottom-up. We start with simple vocabulary concepts, then define terms, atoms and integrity rules. In addition to UML diagrams, which describe the structure of the language, we define the abstract syntax, which is used in Chapter 3 for building the interchange mapping from OCL into SWRL. Section 2.3 specifies the model-theoretic semantics of R2ML, which is needed for proving the semantic correctness of the rule interchange mappings, defined in Chapter 3.

1.3.2 Chapter 3: Rule Interchange between OCL and SWRL

This chapter contains two of the main contributions of the thesis: the mapping from UML/OCL into OWL/SWRL via R2ML and the semantic correctness proof of the mapping (Definition 5).

Section 3.1 gives a motivation for the interchange between OCL and SWRL. The formal problem statement for the interchange task is given in Section 3.2. The following sections of the chapter contain mappings from OCL into SWRL via R2ML and formulate semantic correctness theorems with proofs.

The interchange mapping consists of two mappings: from OCL into R2ML and from R2ML into SWRL.

Section 3.3 defines the mapping from OCL into R2ML. The preliminary Section 3.3.1 specifies the syntax and the semantics of UML class models. The core of the chapter starts in Section 3.3.2, where semantical relations between UML class model and R2ML vocabulary are discussed. Section 3.3.3 defines the mapping from OCL invariants into R2ML integrity rules. The first two preliminary subsections describe a subset of OCL, which can be mapped into R2ML and give an interpretation of OCL expressions (semantics). The following subsections define the compositional mapping from OCL expressions into R2ML and provide the correctness proof of the mapping.

Section 3.4 defines the second mapping: from R2ML into SWRL. The structure of the section is similar to the previous Section 3.3. Preliminary Section 3.4.1 defines syntax and semantics of SWRL. Section 3.4.2 defines semantical relations between R2ML vocabulary and OWL. Section 3.4.3 defines the mapping from R2ML integrity rules into SWRL implications and provides the correctness proof of the mapping.

Section 3.5 discusses limitations of the presented interchange mappings and concludes the chapter.

1.3.3 Chapter 4: Production Rule Verification

This chapter presents a declarative approach to rule verification. In order to verify business rules, we develop so called verifier rules, which, when executed, detect anomalies in business rule bases. We consider Jena Rules as a rule system for expressing and executing business rules, and JBoss Rules for writing verifier rules.

Section 4.1 introduces general verification principles and rule-based (declarative) verification approach. In addition, it presents a rule set with business rules as a running

example for the whole chapter.

Section 4.3 gives an overview of existing works on rule verification.

Preliminary Section 4.4 contains definitions of a knowledge base with production rules and operational semantics of production rules.

A classification and definitions of various anomalies in production rule bases are given in Section 4.5. We use the model theory for defining anomalies and illustrate anomalous situations by means of the running example, given in Section 4.1.

The core of the chapter is in Sections 4.6, where Jena Rules and JBoss Rules production rule systems are introduced first, then a set of verifier rules for anomaly detection is defined. Verifier rules are given in two forms: in semi-formal English language and using the JBoss Rules syntax.

Section 4.6.11 discusses the soundness and completeness of the presented rule-based verification approach.

1.3.4 Chapter 5: Conclusions

This chapter concludes the thesis by discussing relations between rule interchange and rule verification. Possible further research ideas on the basis of these two topics are mentioned.

Chapter 2

R2ML: REWERSE Rule Markup Language

2.1 Introduction and Motivation

In the area of rule modeling there are different developer communities like UML modelers and ontology architects. The former use rules in business modeling and in software development, while the latter use rules in collaborative Web applications. Both of them use different rule languages and tools. Since a business rule is the same rule no matter in which language it is formalized, it is important to support the interchange of rules between different systems and tools.

In this chapter we presents an interchange format for rules integrating the *Rule Markup Language* (RuleML), the *Semantic Web Rule Language* (SWRL) and the *Object Constraint Language* (OCL).

R2ML is a comprehensive and user-friendly XML rule format that allows

- Interchanging rules between different systems and tools;
- Enriching ontologies by rules;
- Connecting your rule system with (our) R2ML-based tools for visualization, verbalization, verification and validation.

R2ML is comprehensive in the sense that it integrates

- The Object Constraint Language (OCL) - a standard used in information systems engineering and software engineering;
- The Semantic Web Rule Language (SWRL) - a proposal to extend the Semantic Web ontology language OWL by adding implication axioms;
- The Rule Markup Language (RuleML) - a proposal based on Datalog/Prolog.

The language includes four rule categories: Derivation rules, production rules, integrity rules and ECA/reaction rules.

R2ML is a *usable* language in the sense that it allows structure-preserving markup and does not force users to translate their rule expressions into a different language paradigm such as having to transform a derivation rule into a FOL axiom, an ECA rule into a production rule, a function into a predicate, or a typed atom into an untyped atom.

Notice that R2ML, like OCL and OWL/SWRL, provides a *rich syntax* for expressing rules supporting conceptual distinctions, e.g. between different types of terms and different types of atoms, which are not present in standard predicate logic. However, a modeler does not have to be familiar with all of R2ML's language elements in order to use it productively.

2.1.1 Existing Rule Languages

In this section we give an overview of various rule languages, which can be considered either as predecessors of R2ML or as a related rule languages for the rule interchange.

Rule Markup Language (RuleML)

Rule Markup Language (RuleML) [15] has been developed by the the Rule Markup Initiative[1] and it permits both forward (bottom-up) and backward (top-down) rules in XML for deduction, rewriting, and further inferential-transformational tasks. RuleML is a pioneer general purpose rule markup language and is widely spread among academic research and has some application prototypes in industry.

RuleML supports different kind of rules: From derivation rules to transformation rules to reaction rules. RuleML can thus specify queries and inferences in Web ontologies, mappings between Web ontologies, and dynamic Web behaviors of workflows, services, and agents.

RuleML is a hierarchy of rule sublanguages upon XML, RDF, XSLT, and OWL. The main sublanguage families of RuleML are *Derivation RuleML*, *PR RuleML* and *Reaction RuleML*.

The model of the Derivation RuleML family of sublanguages is the most developed one and is depicted in Figure 2.1.

The foundation for the kernel of RuleML is the Datalog (constructor-function-free) sublanguage of Horn logic. Datalog is the language in the intersection of SQL and Prolog. It is the subset of logic programming needed for representing the information of null-value-free relational databases, including (recursive) views. That is, in Datalog we can define facts corresponding to explicit rows of relational tables and rules corresponding to tables defined implicitly by views. RuleML Datalog, being a markup language, can conveniently represent relational information where all of the columns are natural-language phrases. A binary datalog subfamily is depicted in blue (Figure 2.1). The negation datalog subfamily is depicted in yellow. The FOL+ subfamily is depicted in green.

The Reaction RuleML initiative[2] has been started recently and incorporated the whole family of RuleML sublanguages for the purpose of rule interchange. The work on Reaction RuleML is still in progress. The data model for production rule actions and the event

[1] RuleML Initiative: http://ruleml.org

[2] Reaction RuleML: http://ibis.in.tum.de/research/ReactionRuleML

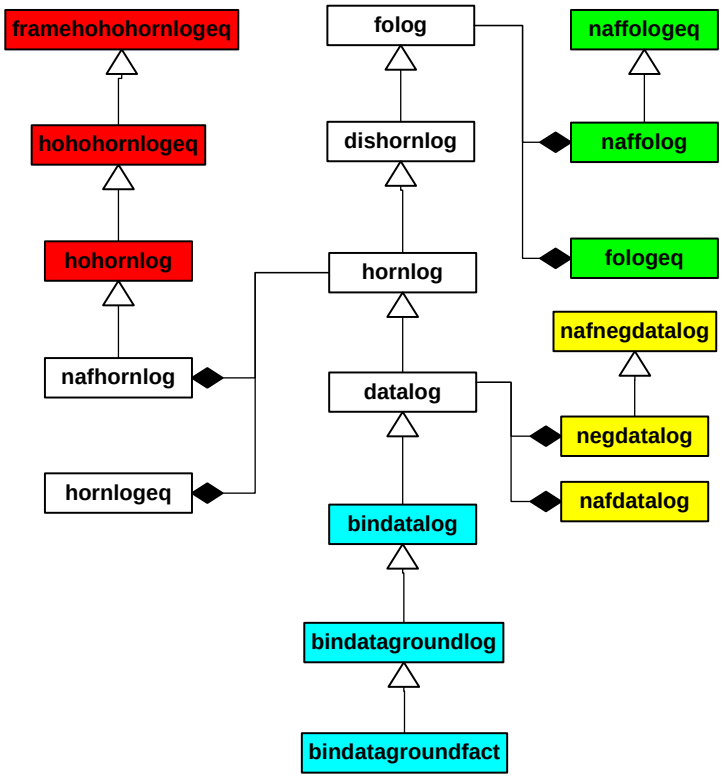

Figure 2.1: The family of Derivation RuleML sublanguages.

model for reaction rules in Reaction RuleML are available on the web site, but these models are not yet well documented, which complicates its usage and analysis. Concerning the practical usage of Reaction RuleML as a rule interchange format, the amount of available translators is currently limited to the interchange from Reaction RuleML into Prova/ContractLog and from Reaction RuleML into Jess.

The issue of interchange between rule languages with complex data types and vocabularies (JBoss Rules, Jena 2) is not well clarified in RuleML.

As a general language for rule interchange, RuleML is not very suitable: as stated in [92], "*in general, a rule interchange will not be loss-free. For instance, since RuleML cannot represent several linguistic distinctions made in OCL and SWRL, an OCL to SWRL interchange is not well supported by RuleML, while R2ML allows a loss-free interchange.*"

Semantic Web Rule Language (SWRL)

The Semantic Web Rule Language (SWRL) [8] is based on a combination of the Web Ontology Language (OWL) with the Unary/Binary Datalog RuleML (see Section 2.1.1 and [15]). SWRL includes a high-level abstract syntax for Horn-like rules in both the OWL DL and OWL Lite sublanguages of OWL. A model-theoretic semantics is given to provide the formal meaning for OWL ontologies including rules written in this abstract syntax ([8], section 3, 5 and 6).

A SWRL rule is an implication between an antecedent (body) and consequent (head). The intended meaning is as following: If the conditions specified in the antecedent hold true, then the conditions specified in the consequent must also hold true.

Both the antecedent and consequent consist of zero or more atoms. An empty antecedent is treated as trivially true (i.e. satisfied by every interpretation), so the consequent must also be satisfied by every interpretation; an empty consequent is treated as trivially false (i.e., not satisfied by any interpretation), so the antecedent must also not be satisfied by any interpretation. Multiple atoms are treated as a conjunction.

There are two concrete syntaxes defined for the SWRL: XML concrete syntax and RDF concrete syntax and an abstract syntax in Extended BNF. As an example of a SWRL rule, consider the following business rule:

> *If a rental is not a one way rental then the return branch of a rental must be the same as the pick-up branch of a rental.*

This rule in informal, human-readable syntax of SWRL:

$$Rental(?x) \wedge \neg OneWayRental(?x) \Longrightarrow returnBranch(?x) = pickupBranch(?x)$$

This rule in SWRL, using Extended BNF abstract syntax:

```
Implies(Antecedent(Rental(I-variable(x))
complementOf(OneWayRental(I-variable(x)))
Consequent(sameAs(returnBranch(I-variable(x)) pickupBranch(I-variable(x))))))
```

The SWRL is the first language proposal specially designed for the Semantic Web. It accommodates concepts of URI and RDF namespaces. SWRL's built-ins approach is based on the reuse of existing built-ins in XQuery and XPath, which are based on XML.

SWRL rules are integrity constraints on top of an OWL ontology. These rules have to be checked against the ontology. Using SWRL rules, a wide class of business rules can be encoded. As a counterpart of OWL/SWRL in the software development community, a tuple UML/OCL can be considered. Rules in OCL are integrity constraints, which are checked against the UML model. Therefore, incorporation of SWRL concepts into general rule markup language, which we define in this chapter is important for rules interchange.

F-Logic

F-Logic is a deductive, object-oriented database language which combines the declarative semantics and expressiveness of deductive database languages with the rich data modeling capabilities supported by the object oriented data model. The basis for a logic programming language which uses objects comes from 1986, when Maier presented his "logic for objects" [53], O-Logic, which was then revisited [42] and finalized [41].

A knowledge representation based on classical logic has some drawbacks such as essentially flat data structures and awkward meta-programming. It is also not very suitable for modeling side effects like state changes and input/output operations. A solution, implemented in F-Logic, includes flat data structures (frames), higher-order syntax (HiLog+F-Logic) and logic of updates (transaction logic). F-Logic provides objects with complex internal structure, class hierarchies and inheritance, typing and encapsulation.

Let's consider the following business rule:

> If a person X is a head of some department where person Y works in, then X is a boss of Y.

Or, in other words: *Someone's boss is the head of that person's department.*

This rule in F-Logic:

$$E[boss \rightarrow M] \leftarrow E{:}Employee\ AND\ D{:}Department\ AND\ E[affiliation \rightarrow D[manager \rightarrow M{:}empl]]$$

The colon denotes a class membership, for example, *E:Employee* specifies that E is a member of class *Employee*. The subclass relationship is denoted by two colons, for instance, *Student is a Person* is *Student::Person*. The complex type (class): $Person[born \Rightarrow integer, name \Rightarrow string, address \Rightarrow string, children \Rightarrow person]$. An object atom is described as $O[Method \rightarrow Value]$, for instance $D[manager \rightarrow M : empl]$, where D is an object, *manager* its attribute and *M:empl* is a value.

F-logic semantics and proof theory is completely general, like that of classical logic. Since FLORA-2 is a programming language based on F-Logic, it uses non-classical semantics: It contains "negation by failure".

F-Logic can be serialized into an XML markup for F-Logic, called F-Logic XML [24]. This format seems to be underdeveloped at the moment and we are not aware of results related to the issue of rules interchange using this format, but F-Logic XML to R2ML translators[3]. However, conceptually, F-Logic is powerful enough to perform rule interchange and widely-known knowledge representation formalism, which is used in real industry applications.

[3]R2ML translators: http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/15

Rule Interchange Format by W3C

The issue of the rule interchange has become important as a huge variety of rule languages and rule systems became available, which lead to the problem of knowledge and rules interoperability. In order to address the problem, the World Wide Web Consortium (W3C) established the Rule Interchange Format Working Group[4] in 2005, which is aimed at releasing a standard for the interchange of rules [78]. If a rule language is compatible with the standard, then rules which are expressed in this language can easily be translated into rules which are expressed in other compatible languages. In other words, the rule interchange format is a rule representation formalism with an XML concrete syntax and a formal semantics. A rule language of a rule system should be compatible with this standard in terms of semantics in order to make their rules interchangable.

The RIF Working Group has released the first working draft of *RIF Core Design* [76] and *RIF Basic Logic Dialect* [75]. At the moment, the main idea of the Working Group is to define a *core*, which is the less general, but extendable rule language with a formal semantics and syntax. The core can be used by rule vendors for the interchange, but due to variety of rule languages, it needs extensions or *dialects* in order to provide loss-free interchange between particular rule languages.

Up to now the first version of the format (RIF) has been released [75]. It defines the syntax and the semantics of rules, but leaves many questions open for the further research. For instance, it does not specify guidelines how to implement a transformation from a source rule language into the RIF. It is also not yet discussed in the RIF Working Group how the correctness of such a translation can be verified. The discussion of the translator's correctness from the semantical point of view is planned by the RIF Working Group.

Related Works on Rule Interchange

Rule interchange is a relatively new research area where little work has been done at the moment. The pioneering results come from the RuleML initiative, already discussed in Section 2.1.1, which proposes the rule markup language RuleML [16] as a rule interchange format.

The research on rule interchange has been conducted in the scope of the REWERSE project. Articles on interchange between SWRL and OCL ([56], [57], [58]) mainly discuss rule transformations and implementation aspects, for instance, the use of Meta Object Facility (MOF) [68] and model transformation techniques for mapping SWRL rules into OCL. The main focus of these works is on implementation of interchange translators. However, these works do not discuss the semantic correctness of the interchange mappings.

In [65] the interchange between rules in JBoss and Jess is discussed. The interchange of production rules is an important issue since these rules are widely used in the industry. In this work, R2ML is used as an interchange format.

There is research on the interchange of rules, which express security policies ([36], [37], [38]). The authors mainly analyze how policies expressed in different rule languages can be interchanged via R2ML. The issue of the semantic correctness is also left beyond the

[4]RIF WG: `http://www.w3.org/2005/rules/wiki/RIF_Working_Group`

discussion.

The R2ML XML is used as a serialization format for URML rules [93]. The rule modeling tool "Strelka" generates R2ML XML from rule models and supports further translation to such rule languages as Jena, JBoss Rules and others [50].

Another work on modeling Semantic Web rules by means of UML is presented in [18]. It defines the UML profile for rule-extended OWL DL ontologies and the Semantic Web Rule Language (SWRL). However, this work is generalized in [19], where the MOF-based metamodeling framework is presented for development of Semantic Web ontologies and rules. The OCL constraints in this framework are used only for improving the precision of metamodels and their mappings into Semantic Web rules is not considered. The goal of this metamodeling approach is to facilitate the adoption of semantic technologies in real-life applications. Our mappings from OCL to SWRL, defined in this thesis, have the same goal of promoting the use of UML technologies for modeling Semantic Web rules. However, our approach focuses on rule interchange via an intermediate rule representation, while the metamodeling approach by Brockmans is mainly related to the modeling of ontologies and rules.

We may conclude that up to now there are many activities in the area of rule interchange. However, there are still a lot of unresolved questions, in particular concerning the rule interchange mappings and their semantic correctness. The goal of this work is to contribute to the work on these open questions and give a formal correctness proof of the existing and forthcoming rule translators.

R2ML versus RIF as an Interchange Language for Rules

We employ R2ML for our rule interchange research, which is presented in Chapter 3. The main argument in favor of R2ML over W3C RIF is that at the moment, when current research has been conducted, the RIF was not a standard, but still work in progress, while R2ML has already been evaluated in various rule applications and case studies [52]. Therefore, due to historical and practical reasons R2ML was the only acceptable choice as an interchange format.

Concerning relations between R2ML and RIF, we say that R2ML can be considered as a possible RIF dialect with the reacher syntax. The semantics compatibility between R2ML and RIF has not been investigated since the work on RIF semantics was not finalized until the time of the current research. However, the issue of R2ML compatibility with W3C RIF has been investigated in [49]. In summary, R2ML is compatible with RIF goals and RIF requirements and use cases.

The same document [49] analyzes R2ML compatibility with RIF BLD from the semantical and syntactical points of view. In general, R2ML does not provide specific semantics. As it is stated in [49], "*R2ML (as well as RIF), as an interchange language must allow interchanging rules from different platforms therefore should accommodate the semantics of the target languages.*" In this work on rule interchange between OCL and SWRL we define R2ML semantics which accommodate semantics of these target languages.

It is shown that RIF syntax is more general. Because of that, "*a translation from R2ML to RIF is straightforward, however, translation from R2ML to RIF is performed by loosing information which was previously encoded in R2ML. Therefore an inverse translation from*

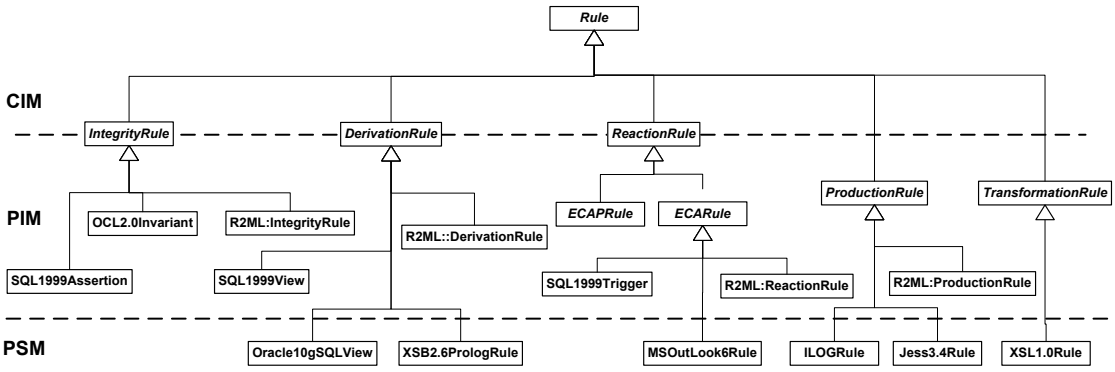

Figure 2.2: Rule concepts at three different abstraction levels: computation-independent (CIM), platform-independent (PIM) and platform-specific (PSM) modeling (Classification by Gerd Wagner from [6]).

RIF to R2ML will not preserve the initial R2ML markup."

Finally, the formal problem statement, defined in Section 3.2, does not depend on a particular rule interchange format. Any language, which satisfies rule interchange format requirements [78] can be used for implementing interchange mappings.

2.1.2 Methodology

We adopt the Model Driven Architecture (MDA,[68]), which is a framework for distinguishing different abstraction levels defined by the Object Management Group (OMG). As illustrated in Figure 2.2, we consider rules at the three abstraction levels defined be the MDA:

At the **("computation-independent") business domain level** (called CIM in OMG's MDA), rules are statements that express (certain parts of) a business/domain policy (e.g., defining terms of the domain language or defining/constraining domain operations) in a declarative manner, typically using a natural language or a visual language. Examples of such rules are:

- *"The driver of a rental car must be at least 25 years old."*
- *"A gold customer is a customer with more than $1Million on deposit."*
- *"An investment is exempt from tax on profit if the stocks have been bought more than a year ago."*
- *"When a share price drops by more than 5% and the investment is exempt from tax on profit, then sell it."*

At the **platform-independent operational design level** (called PIM in OMG's MDA), rules are formal statements, expressed in some formalism or computational paradigm, which can be directly mapped to executable statements of a software system. Examples of rule languages at this level are SQL:1999 [86], OCL 2.0 [4] and DOM Level 3 Event

Listeners [2]. Remarkably, SQL provides operational constructs for all three business rule categories mentioned above: Checks and assertions operationalize a notion of integrity rules, views operationalize a notion of derivation rules, and triggers operationalize a notion of reaction rules.

At the **platform-specific implementation level** (called PSM in OMG's MDA), rules are statements in a language of a specific execution environment, such as Oracle 10g views [70], Jess 3.4 [35], XSB 2.6 Prolog [95] or the Microsoft Outlook 6 Rule Wizard [71].

Rule interchange will be important both at the PIM and the PSM level. So, there are four interchange types:

PIM to PIM examples: OCL to SQL, SQL to ISO Prolog.

PIM to/from PSM examples: OCL to/from Java, SQL to/from Oracle 10g.

PSM to PSM examples: XSB Prolog to SWI Prolog, ILOG to ILOG, ILOG to Jess.

General purpose rule interchange formats, such as RuleML and R2ML, address the PIM level. They support a PSM to PSM interchange via the PIM level. Since there will be several rule interchange formats, there is also the issue of mapping them on each other.

In general, a rule interchange will not be loss-free. For instance, since RuleML cannot represent several linguistic distinctions made in OCL and SWRL, an OCL to SWRL interchange is not well supported by RuleML, while R2ML allows a loss-free interchange.

We assume that Web rule languages do not directly follow the tradition of predicate-logic-style rule languages such as Prolog, but rather follow the recent developments of Web knowledge representation languages such as RDF [7] and OWL [5]. This requires that they accommodate:

- *Web naming concepts*, such as URIs/IRIs and XML namespaces,
- *The ontological distinction between objects and data values*,
- *The datatype concepts of RDF*.

As a working name for our interchange format, we use the acronym R2ML standing for *REWERSE Rule Markup Language*.

The summary of R2ML design principles is as following:

- Modeled using MDA;
- Rule concepts defined with the help of MOF/UML;
- Actions (following OMG PRR submission);
- EBNF abstract syntax;
- XML based concrete syntax validated by an XML Schema;
- Allowing different semantics for rules.

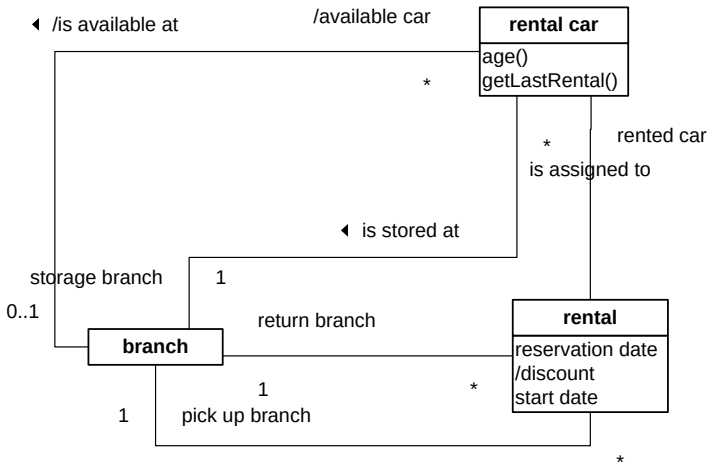

Figure 2.3: A sample business vocabulary of EU-Rent case study.

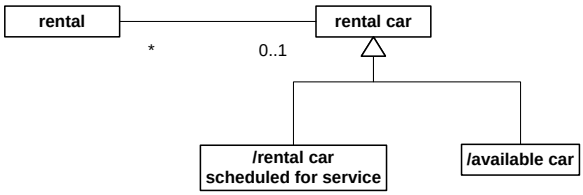

Figure 2.4: A sample business vocabulary of EU-Rent case study.

2.2 R2ML Metamodel

We define R2ML incrementally, starting from basic vocabulary concepts like classes and properties and advancing to rule constituents like terms and atoms. In order to illustrate language concepts defined in following sections by means of metamodelling, we refer to two UML class diagrams, depicted in Figure 2.3 and Figure 2.4. These diagrams represents a part of a business vocabulary of the EU-Rent case study, which is recommended by the Business Rule Conference[5] for an evaluation of rule methodologies and examples.

Figure 2.4 depicts classes *rental car*, *branch* and *rental*. Every branch has a rental car as available car. Every rental car *is available at* at most one branch. Every rental car *is stored at* exactly one branch as a *storage branch*. Each rental car *is assigned to* a rental. Every rental has a rental car as a *rented car*. Every rental has exactly one branch as a *pick up branch* and exactly one branch as a *return branch*.

Figure 2.3 depicts a class *rental car*, which is either a *rental car scheduled for service* or an *available car*. These two subclasses of a rental car class are derived concepts, which means that they are defined by means of business rules. A rental car is assigned to a *rental*.

[5]Business Rule Conference

2.2.1 Basic Content Vocabulary

The basic user-defined content vocabulary (see Figure 2.5) consists of vocabulary entries and each entry may include:

- user-defined **noun concept names** standing for general noun concepts, denoted by a *type*, which is either a *class* (*object type*) or a *datatype*. For instance, *rental* and *rental car* are classes (Figure 2.4).

- user-defined **verb concept names**, called "predicate symbols" in traditional logic, referring to general verb concepts, or *predicates*. A predicate is either a *property*, *datatype predicate* or an *association predicate*. A property is either an *attribute*, if it is data-valued (range role is a datatype) or a *reference property*, if it is object-valued (range role is a class). An R2ML attribute is a special type of user-defined function that corresponds to a datavalued property in a UML class model. In Figure 2.3, `reservationDate` is an attribute of the class Rental. A reference property corresponds to a functional association end (of a binary association) in a UML class model. For instance in Figure 2.3, both association ends `pickupBranch` and `returnBranch` define reference properties;

- user-defined **object names**;

In Web languages such as RDF [7] and OWL [5], all these names are globally unique standard identifiers in the form of URI references. One of the goals of R2ML is to comply with important Semantic Web standards like RDF(S) and OWL. In particular, R2ML accommodates the datatype concept of RDF.

User-defined noun concepts comprise **classes** (or *object types*). Usually, any object or object variable belongs to a class. A *class* in R2ML is denoted by a URI reference. A class is a type entity (or classifier) for R2ML objects and object variables.

The datatype language consists of a set of predefined datatype names, including the name `rdfs:Literal` referring to the generic built-in datatype, the lexical space of which is the set of all Unicode strings. Each predefined datatype name is associated with:

- a set of **data literals**, which are Unicode strings and are either *typed literals* or *plain literals*;

- a set of **datatype function names**;

- a set of **datatype predicate names**.

The datatype concept of R2ML is an extension of the datatype concept of RDF. Each datatype is denoted by an URI reference. A datatype is a type for R2ML data values and data variables.

Notice that we use an attribute name both as the name of a function and as the name of the corresponding functional predicate. Likewise, we use a reference property name both as the name of a property predicate and as the name of the corresponding role function. This kind of naming liberty, which is supported by RDF [7] and Common Logic[21], helps to switch between functional and relational languages.

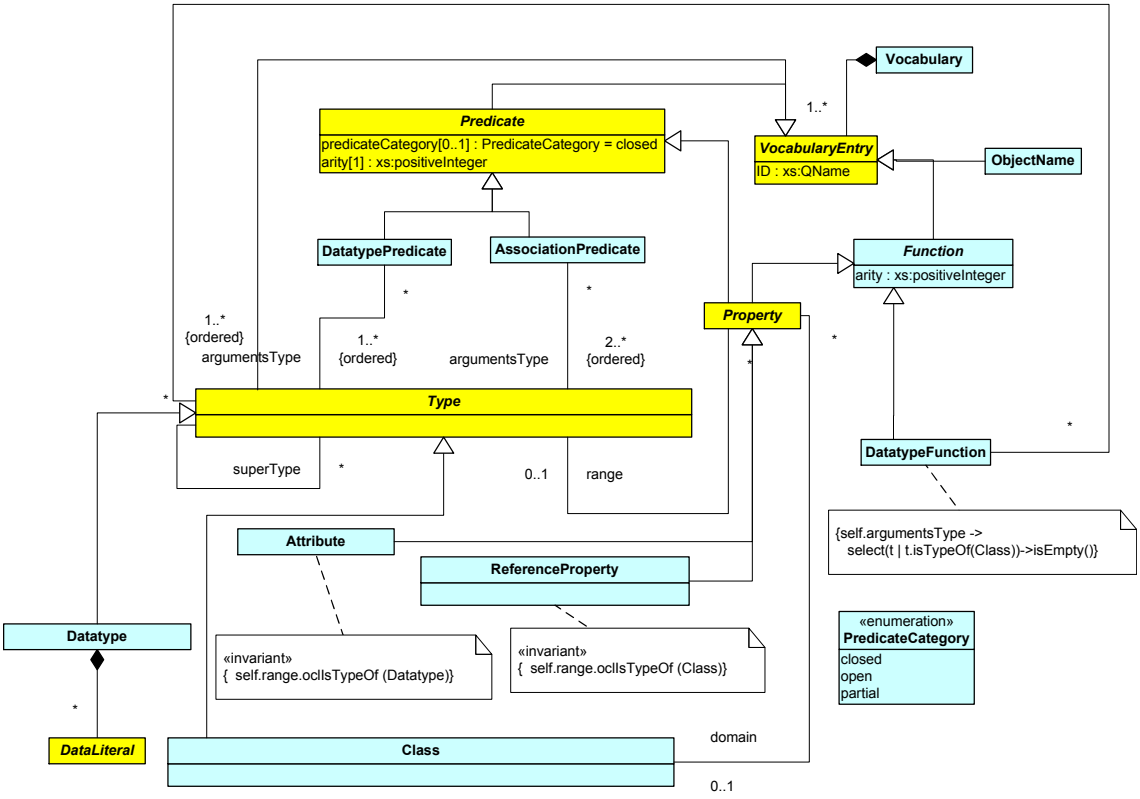

Figure 2.5: R2ML vocabulary metamodel

In R2ML, a *reference property* is denoted by a URI reference. It corresponds to a non-literal-valued RDF property or to an OWL 'object property' and it is used in R2ML reference property atoms (see Section 2.2.3).

A *datatype predicate* in R2ML is denoted by a URI reference. The R2ML datatype predicate accommodates the SWRL concept of a *built-in predicate*.

An R2ML *function* is a user-defined function that corresponds to the standard logic concept of function (see Figure 2.5). It is either a *datatype function* or an *operation*. Since OWL/SWRL does not support user-defined operations, we do not consider user-defined operations in R2ML in this work.

An *association predicate* in R2ML is denoted by a URI reference. It allows capturing n-ary associations and association classes. Association predicates are used in association predicate atoms (see Section 2.2.3). For instance, an association rental car is assigned to a rental in Figure 2.4 represents an association predicate.

2.2.2 Terms

The general structure of R2ML terms is depicted in Figure 2.6.

Typed terms are either *object terms* standing for *objects*, or *data terms* standing for *data values*. The concrete syntax of first-order non-boolean OCL expressions can be

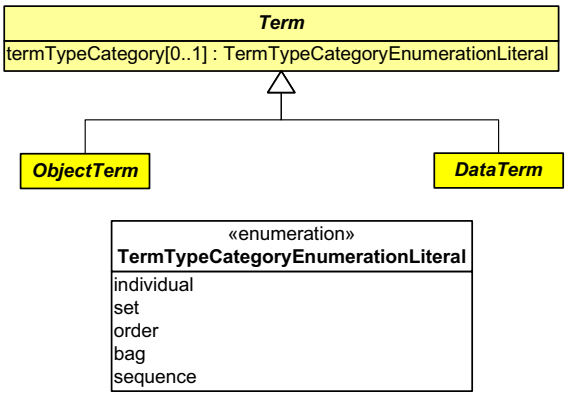

Figure 2.6: R2ML terms

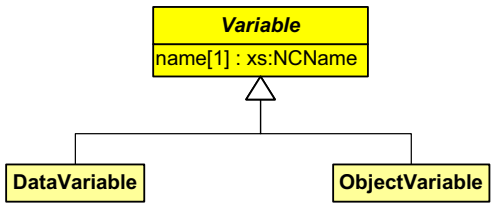

Figure 2.7: R2ML variable

directly mapped onto our abstract concepts of *ObjectTerm* and *DataTerm*, which can be viewed as a predicate-logic-based reconstruction of the standard OCL [4] abstract syntax.

R2ML provides the concept of a *Variable* (see Figure 2.7), but as in many programming languages, distinguishes between *object* and *data* variables, i.e between references that are instantiated with objects or with data values.

An *object term* (see Figure 2.8) is either an *object variable*, an *object name*, or a *reference property function term*, which corresponds to a functional association end (of a binary association) in a UML class model.

A *data term* is either a *data variable*, a *data literal*, or a *data function term*, which can have two different types:

1. A *datatype function term* formed with the help of a datatype function that comes with the corresponding datatype.

2. An *attribute function term* formed with the help of a user-defined attribute.

The expression `x.pickupBranch`, where *pickupBranch* is a role name of a *Branch* in association between classes *Rental* and *Branch* (see Figure 2.3), is a reference property function term, where `x` is an object term and `pickupBranch` is a reference property.

Below comes the R2ML abstract syntax, which is used in Chapter 3 on interchange between OCL and SWRL:

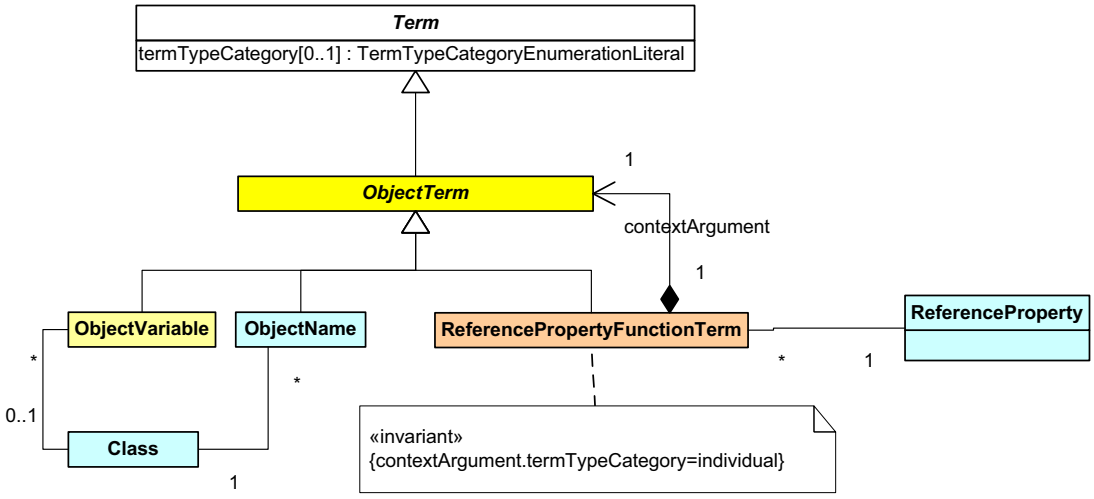

Figure 2.8: R2ML object term

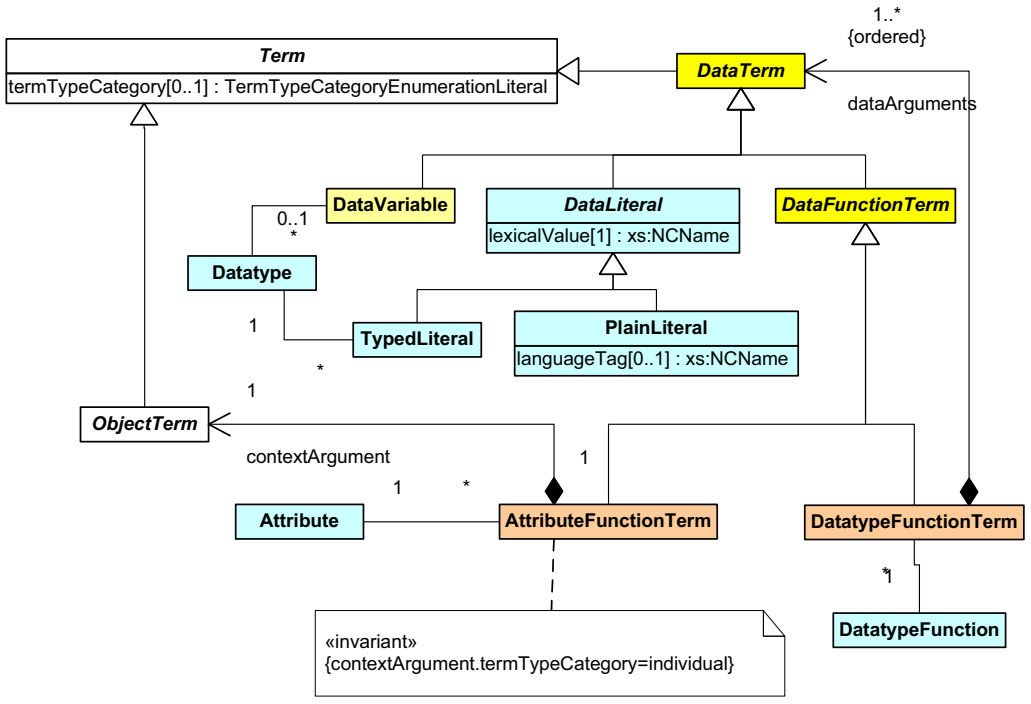

Figure 2.9: R2ML data term.

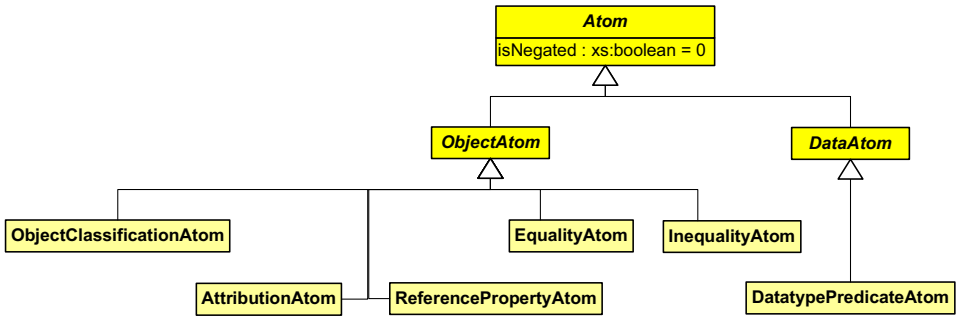

Figure 2.10: R2ML atoms

⟨*Term*⟩ **::=** ⟨*ObjectTerm*⟩|⟨*DataTerm*⟩.

⟨*Variable*⟩ **::=** ⟨*DataVariable*⟩|⟨*ObjectVariable*⟩.

⟨*ObjectTerm*⟩ **::=** ⟨*ObjectVariable*⟩|⟨*ObjectName*⟩|⟨*ReferencePropertyFunctionTerm*⟩.

⟨*ObjectVariable*⟩ **::= OVar(**⟨*name*⟩**, class(**[⟨*classID*⟩]**)).**

⟨*ObjectName*⟩ **::= ObjN(**⟨*objNID*⟩**, class(**[⟨*classID*⟩]**)).**

⟨*ReferencePropertyFunctionTerm*⟩ **= RefPropFuncTerm(**⟨*refPropID*⟩**, cxtArg(**⟨*ObjectTerm*⟩**)).**

⟨*DataTerm*⟩ **::=** ⟨*DataVariable*⟩|⟨*DataLiteral*⟩|⟨*DataFunctionTerm*⟩.

⟨*DataVariable*⟩ **::= DVar(**⟨*name*⟩**, dType(**[⟨*dTypeID*⟩]**)).**

⟨*DataLiteral*⟩ **::= DLit(**⟨*lexicalValue*⟩**).**

⟨*DataFunctionTerm*⟩ **::=** ⟨*AttributeFunctionTerm*⟩|⟨*DatatypeFunctionTerm*⟩.

⟨*AttributeFunctionTerm*⟩ **::= AttrFuncTerm(**⟨*attrID*⟩**, cxtArg(**⟨*ObjectTerm*⟩**),**
args(⟨*Term*⟩**, {**⟨*Term*⟩**})).**

⟨*DatatypeFunctionTerm*⟩ **::= DatatypeFuncTerm(**⟨*dTFunID*⟩**, dataArgs(**⟨*DataTerm*⟩**, {**⟨*DataTerm*⟩**})).**

2.2.3 Atoms

The basic constituent of a rule is an *atom*. In R2ML we define atoms, which are compatible with all important concepts of OWL, SWRL and RuleML.

An atom is either an object atom or a data atom (see Figure 2.10). In R2ML we define atoms, which are compatible with all important concepts of OWL, SWRL and RuleML.

An *object classification atom* (see Figure 2.11) refers to a class and consists of an object term. The purpose of an object classification atom is to classify an object term.

An *attribution atom* (see Figure 2.12) consists of an attribute, an object term as "subject", and a data term as "value".

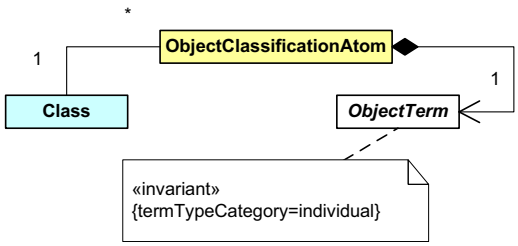

Figure 2.11: R2ML object classification atom.

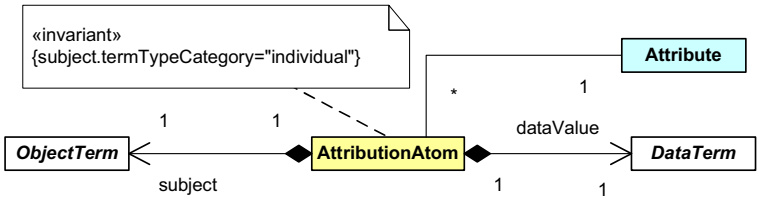

Figure 2.12: R2ML attribution atom

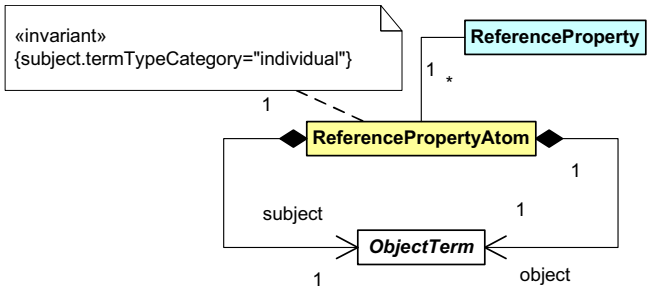

Figure 2.13: R2ML reference property atom

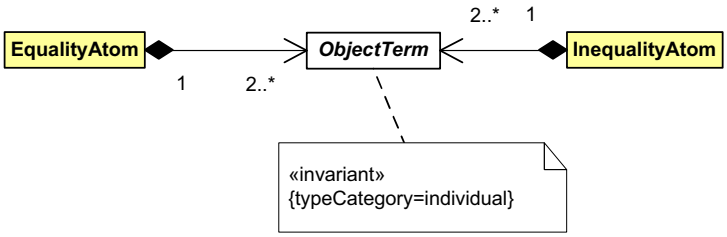

Figure 2.14: R2ML equality and inequality atoms.

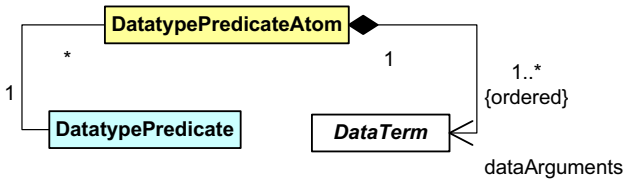

Figure 2.15: R2ML datatype predicate atom.

A *reference property atom* associates an object term as "subject" with other object term as "object". This atom corresponds to the concept of an RDF triple with a non-literal object.

As in SWRL [8], R2ML supports concepts of equality and inequality atoms. An R2ML *equality atom* or *inequality atom* is composed of two or more object terms.

An R2ML *data predicate atom* refers to a datatype predicate, and consists of a number of data terms as data arguments.

Below comes the R2ML abstract syntax:

⟨*Atom*⟩ **::=** ⟨*ObjectAtom*⟩|⟨*DataAtom*⟩**.**

⟨*ObjectAtom*⟩ **::=** ⟨*ObjectClassificationAtom*⟩
|⟨*AttributionAtom*⟩
|⟨*ReferencePropertyAtom*⟩
|⟨*EqualityAtom*⟩
|⟨*InequalityAtom*⟩**.**

⟨*ObjectClassificationAtom*⟩ **::= ObjClAt(class(**⟨*classID*⟩**), term(**⟨*ObjectTerm*⟩**)).**

⟨*AttributionAtom*⟩ **::= AttrAt(**⟨*attrID*⟩**, subj(**⟨*ObjectTerm*⟩**), dataVal(**⟨*DataTerm*⟩**)).**

⟨*ReferencePropertyAtom*⟩ **::= RefPropAt(**⟨*refPropID*⟩**, subj(**⟨*ObjectTerm*⟩**), obj(**⟨*ObjectTerm*⟩**)).**

⟨*EqualityAtom*⟩ **::= EqAt(**⟨*ObjectTerm*⟩**,** ⟨*ObjectTerm*⟩**).**

⟨*InequalityAtom*⟩ **::= IneqAt(**⟨*ObjectTerm*⟩**,** ⟨*ObjectTerm*⟩**).**

⟨*DatatypePredicateAtom*⟩ **::= DPredAt(**⟨*dTPredID*⟩**, dataArgs(**⟨*DataTerm*⟩**, {**⟨*DataTerm*⟩**})).**

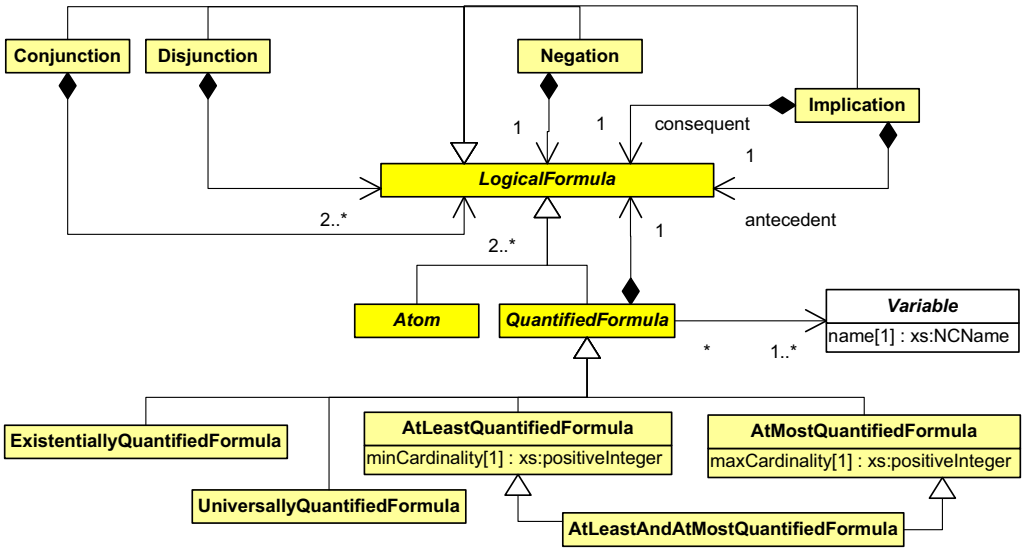

Figure 2.16: R2ML logical formula

2.2.4 Formulae

R2ML provides the concept of *LogicalFormula*, which corresponds to a general first order formula and instantiates in quantified formulas that are existentially quantified formulas or universally quantified formulas. The concept of a quantified formula is mandatory for R2ML integrity rules.

Below comes the R2ML abstract syntax:

⟨*LogicalFormula*⟩ **::=** ⟨*Atom*⟩
| ⟨*Conjunction*⟩
| ⟨*Disjunction*⟩
| ⟨*Implication*⟩
| ⟨*QuantifiedFormula*⟩
| ⟨*Negation*⟩**.**

⟨*Conjunction*⟩ **::= And(**⟨*LogicalFormula*⟩**,** ⟨*LogicalFormula*⟩**,**
{⟨*LogicalFormula*⟩**}**
).

⟨*Implication*⟩ **::= Impl(cons(**⟨*LogicalFormula*⟩**), antec(**⟨*LogicalFormula*⟩**)).**

⟨*QuantifiedFormula*⟩ **::=** ⟨*ExistentiallyQuantifiedFormula*⟩ |
⟨*UniversallyQuantifiedFormula*⟩|
⟨*AtLeastQuantifiedFormula*⟩|
⟨*AtMostQuantifiedFormula*⟩|
⟨*AtLeastAndAtMostQuantifiedFormula*⟩**.**

⟨*ExistentiallyQuantifiedFormula*⟩ **::= ExQuantF(vars(**⟨*Variable*⟩**, {**⟨*Variable*⟩**}),**
⟨*LogicalFormula*⟩
).

⟨*UniversallyQuantifiedFormula*⟩ **::= UnivQuantF(vars(**⟨*Variable*⟩**, {**⟨*Variable*⟩**}),**
⟨*LogicalFormula*⟩
).

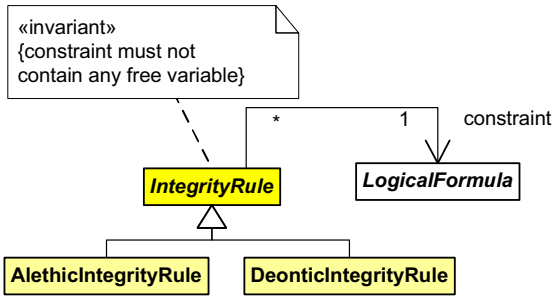

Figure 2.17: R2ML integrity rule

⟨*AtLeastQuantifiedFormula*⟩ **::= AtLeastQuanF(vars(**⟨*Variable*⟩**, {**⟨*Variable*⟩**}),**
⟨*LogicalFormula*⟩**, minCard(**⟨*minCardinality*⟩**)**
).

⟨*AtMostQuantifiedFormula*⟩ **::= AtMostQuanF(vars(**⟨*Variable*⟩**, {**⟨*Variable*⟩**}),**
⟨*LogicalFormula*⟩**, maxCard(**⟨*maxCardinality*⟩**)**
).

⟨*Negation*⟩ **::= Neg(**⟨*LogicalFormula*⟩**).**

⟨*minCardinality*⟩ **::=** ⟨*xs:positiveInteger*⟩**.**

2.2.5 Integrity Rule

An integrity rule, also known as (an integrity) constraint (see Figure 2.17), consists of a constraint assertion, which is a sentence in a logical language such as first-order predicate logic or OCL. R2ML framework supports two kind of integrity rules: The *alethic* and the *deontic* ones. The alethic integrity rule can be expressed by a phrase, such as "it is necessarily the case that". The deontic one can be expressed by phrases, such as "it is obligatory that" or "it should be the case that". The corresponding *LogicalFormula* of an integrity rule must have no free variables, i.e. all the variables from this formula must be quantified. In this paper we use only alethic integrity rules.

⟨*AlethicIntegrityRule*⟩ **::= AIR(constr(**⟨*LogicalFormula*⟩**)).**

For the full R2ML abstract syntax we refer to the R2ML homepage (`http://rewerse.net/I1`).

2.3 Direct Model-Theoretic Semantics of R2ML

The meaning of rules, expressed in some rule language is what they state about the world. The rule sentences do not have meaning by themselves. In order to establish correspondence between rule sentences and facts, the rule language must have an *interpretation*. One approach for defining semantics by means of interpretations is called *model theory*. The main goal of a formal semantic theory is to provide the technical means for determining when inference processes are valid, i.e. when they preserve truth. An interpretation

provides enough information about a state of a "possible world", fixing the truth-value of any ground R2ML atom.

A similar approach has been used for defining the semantics of RDF [7] and OWL [5]. The R2ML semantics in this section is based on [31], where the direct model-theoretic semantics for NafNeg-Ur-Datalog sublanguage of RuleML have been introduced. We extend this work to the condition language of R2ML, taking into consideration advanced R2ML features like different kinds of logical formulas and properties.

Definition 1 (R2ML Vocabulary)**.** *An* R2ML vocabulary *is defined by the following tuple*

$$\mathcal{V}oc = (\text{Pred}, \text{Fun}, \text{Prop}, \text{DLit}, \text{Obj})$$

where

1. Pred *is the set of association predicates (*OPred*), datatype predicates (*DPred*) and properties (*Prop*);*
2. Fun *is the set of object function symbols (*OFun*) and data function symbols (*DFun*);*
3. Prop *is the set of reference properties (*OProp*) and attributes (*DProp*);*
4. DLit *is the set of all data literals;*
5. Obj *is the set of object names.*

Let $\mathcal{O}$ be a set of all objects and $\mathcal{D}$ be a set of all data values. In this work each predicate is interpreted as $p \in (\mathcal{O} \cup \mathcal{D})^n$.

Definition 2 (Interpretation)**.** *An abstract interpretation is a tuple of functions*

$$\mathcal{I} = (\mathcal{I}_{DLit}, \mathcal{I}_{Obj}, \mathcal{I}_{Fun}, \mathcal{I}_{Pred}, \mathcal{I}^{fun}_{Prop}, \mathcal{I}^{rel}_{Prop})$$

such that:

1. $\mathcal{I}_{DLit} : \text{DLit} \longrightarrow \mathcal{D}$*, maps each data literal dl into a value*

$$\mathcal{I}(dl) = \mathcal{I}_{DLit}(dl) \in \mathcal{D}$$

2. $\mathcal{I}_{Obj} : \text{Obj} \longrightarrow \mathcal{O}$*, maps each object name oid to an object*

$$\mathcal{I}(oid) = \mathcal{I}_{Obj}(oid) \in \mathcal{O}$$

3. $\mathcal{I}_{Fun}$ *maps each n-ary function symbol* $f \in Fun$ *into an n-ary function* $\mathcal{I}_{Fun}(f) \in (\mathcal{O} \cup \mathcal{D})^n \longrightarrow (\mathcal{O} \cup \mathcal{D})$*;*
   (a) $\mathcal{I}_{OFun}$ *maps each n-ary object function symbol* $f \in OFun$ *into an n-ary object function* $\mathcal{I}_{OFun}(f) \in (\mathcal{O} \cup \mathcal{D})^n \longrightarrow \mathcal{O}$*;*
   (b) $\mathcal{I}_{DFun}$ *maps each n-ary data function symbol* $f \in DFun$ *into an n-ary data function* $\mathcal{I}_{DFun}(f) \in (\mathcal{O} \cup \mathcal{D})^n \longrightarrow \mathcal{D}$*;*

4. $\mathcal{I}_{Pred}$ *maps each n-ary predicate* $p \in Pred$ *into an n-ary relation* $\mathcal{I}_{Pred}(p) \in (\mathcal{O} \cup \mathcal{D})^n$;

   (a) $\mathcal{I}_{OPred}$ *maps each object predicate* $p \in OPred$ *into an n-ary relation* $\mathcal{I}_{OPred}(p) \in \mathcal{O} \times (\mathcal{O} \cup \mathcal{D})^{n-1}$;

   (b) $\mathcal{I}_{DPred}$ *maps each datatype predicate* $p \in DPred$ *into an n-ary datatype relation* $\mathcal{I}_{DPred}(p) \in \mathcal{D}^n$;

5. $\mathcal{I}^{fun}_{Prop}$ *maps each property* $p \in Prop$ *into a function* $\mathcal{I}^{fun}_{Prop}(p) \in \mathcal{O} \longrightarrow \mathcal{O} \cup \mathcal{D}$;

   (a) $\mathcal{I}^{fun}_{OProp}$ *maps each reference property* $p \in OProp$ *into an object function* $\mathcal{I}^{fun}_{OProp}(p) \in \mathcal{O} \longrightarrow \mathcal{O}$;

   (b) $\mathcal{I}^{fun}_{DProp}$ *maps each attribute* $p \in DProp$ *into a datatype function* $\mathcal{I}^{fun}_{DProp}(p) \in \mathcal{O} \longrightarrow \mathcal{D}$;

6. $\mathcal{I}^{rel}_{Prop}$ *maps each property* $p \in Prop$ *into a binary relation* $\mathcal{I}^{rel}_{Prop}(p) \in \mathcal{O} \times \mathcal{O} \cup \mathcal{D}$;

   (a) $\mathcal{I}^{rel}_{OProp}$ *maps each reference property* $p \in OProp$ *into a binary object relation* $\mathcal{I}^{rel}_{OProp}(p) \in \mathcal{O} \times \mathcal{O}$;

   (b) $\mathcal{I}^{rel}_{DProp}$ *maps each attribute* $p \in DProp$ *into a binary datatype relation* $\mathcal{I}^{rel}_{DProp}(p) \in (\mathcal{O} \times \mathcal{D})^2$;

Definition 3 (Valuation)**.** *A valuation over an interpretation* $\mathcal{I}$ *is a function* $\mathcal{V} : \mathrm{Var} \longrightarrow \mathcal{O} \cup \mathcal{D}$*, which associates each variable name* v *with a value, i.e.*

1. *If* $v \in \mathcal{O}Var$*, then* $\mathcal{I}(v) = \mathcal{V}(v) \in \mathcal{O}$
2. *If* $v \in \mathcal{D}Var$*, then* $\mathcal{I}(v) = \mathcal{V}(v) \in \mathcal{D}$

Definition 4 (Satisfaction Relation)**.** *Let* $\mathcal{V}$ *be a valuation. The* satisfaction relation $\models$ *is such that,*

1. *Let* p *be an n-ary predicate,* $\mathcal{I}_{Pred}(r) = (p)$*, where* $p \in (\mathcal{O} \cup \mathcal{D})^n$ *and* A *is an atom. Then*
   $\mathcal{I},\mathcal{V} \models A(p, t_1, \ldots, t_n)$ *iff* $\langle \mathcal{I}(t_1), \ldots, \mathcal{I}(t_n) \rangle \in p$
2. *Let* $And(A_1, ..., A_n)$ *be a conjunction of atoms, then*
   $\mathcal{I},\mathcal{V} \models And(A_1, ..., A_n)$ *iff for all* i*,* $1 \leq i \leq n$*,* $\mathcal{I},\mathcal{V} \models A_i$
3. *Let* $Or(A_1, ..., A_n)$ *be a disjunction of atoms, then* $\mathcal{I},\mathcal{V} \models Or(A_1, ..., A_n)$ *iff exists* i*,* $1 \leq i \leq n$*,* $\mathcal{I},\mathcal{V} \models A_i$
4. *Let* $Neg(A)$ *be a strong negation of atom* A*, then* $\mathcal{I},\mathcal{V} \models Neg(A)$ *iff* $\mathcal{I},\mathcal{V} \not\models A$
5. *Let* $Impl(cons(lf_1), antec(lf_2))$ *be a material implication, where* lf_1 *and* lf_2 *are logical formulas, then*
   $\mathcal{I},\mathcal{V} \models Impl(cons(lf_1), antec(lf_2))$ *iff* $\mathcal{I},\mathcal{V} \models lf_1$ *or not* $\mathcal{I},\mathcal{V} \models lf_2$

6. *Let $F(x)$ be an at least (n) and at most (m) quantified formula, then*

   (a) *Exists $S = \{a_1, \ldots, a_k\}$ such that $n \leq |S| \leq m$ and $\mathcal{I}_{\mathcal{V}x/b} \models F(x)$ iff $b \in S$;*
   $\mathcal{I},\mathcal{V} \models \exists^{[n,m]} x|F(x)$ iff exists $a_1, \ldots, a_k, n \leq k \leq m$ such that for all $i = 1 \ldots k$,
   $\mathcal{I},\mathcal{V}_{x/a_i} \models F(x)$;

7. *Let $F(x)$ be an existentially quantified formula, then*
   $\mathcal{I},\mathcal{V} \models \exists x|F(x)$ iff exists $a \in \mathcal{O} \cup \mathcal{D}$ such that $\mathcal{I}_{\mathcal{V}x/a} \models F(x)$;

8. *Let $F(x)$ be a universally quantified formula, then*
   $\mathcal{I},\mathcal{V} \models \forall x|F(x)$ iff for all a from the domain $\mathcal{I}_{\mathcal{V}x/a} \models F(x)$;

2.4 On the Theoretical Properties of R2ML

The theoretical properties of R2ML, such as computability and complexity, have not been investigated explicitly since it was not the goal of the research on rule interchange. The theoretical properties are crucial if there is a need for building a rule engine and performing reasoning. But R2ML is supposed to be used as an intermediate representation for rules and not as a formalism for reasoning. An important R2ML design goal which addresses the loss-free interchange problem, is the rich syntax and tolerance for different semantics for rules.

On the other hand, R2ML can be used as a concrete XML syntax for Extended RDF ([12], [11]), which extends the RDF(S) semantics and is based on Partial Logic. The ERDF framework supports both closed-world and open-world reasoning through the explicit representation of the particular closed-world assumptions and the ERDF ontological categories of total properties and total classes.

Theoretical properties of ERDF are investigated in [13]. The work provides a modified semantics for ERDF ontologies, called ERDF ♯n-stable model semantics. It is shown, that entailment under this semantics is in general decidable.

A subset of R2ML, which is used for performing the interchange between OCL and SWRL, can be mapped into ERDF, therefore, ERDF theoretical properties are applicable to this subset of R2ML.

Chapter 3

Rule Interchange between OCL and SWRL

3.1 Introduction and Motivation

Business rules are intensively used in business modeling and requirements engineering. Being initially expressed in the natural language, they can be formalized using different (formal) rule languages.

The Unified Modeling Language (UML) [67] is a mainstream modeling technology in software engineering. Software developers capture business rules, expressed by business experts in the natural language, and usually specify them using the Object Constraint Language (OCL) [4], which has a formal semantics and is designed to express rules on top of UML class diagrams using textual syntax.

As it is stated in [92], "since a business rule is the same rule no matter in which language it is formalized, it is important to support the interchange of rules between different systems and tools." A particular business scenario is when a company has rules in one rule language and then needs to deploy them into a rule system with another rule language. For instance, a software engineer can formalize rules by means of OCL, but later comes the need to have the rules in the Semantic Web Rule Language (SWRL) [8] in order to use them in a Semantic Web application. In this case the developer has to solve the *rule interchange problem*. This chapter focuses on the rule interchange between OCL and SWRL. In particular, we define a rule mapping from OCL into SWRL, which brings the following impact:

- It bridges the gap between two modeling communities: Software developers, who mainly use UML/OCL and ontology architects, who deal with Semantic Web standards as OWL and SWRL. Rules interoperability between OCL and SWRL allows employing UML/OCL engineers as Semantic Web rule architects.
- It facilitates the growth of the Semantic Web by employing existing UML technologies and tools for modeling ontologies and rules for the Semantic Web.

OCL supports various rule types: Invariants, also known as (integrity) constraints and derivation rules, which are used to define data values of attributes and object values of

association ends. In this work we define a mapping from OCL invariants into SWRL rules. We focus on OCL invariants since they are widely used for formalization of various business rules in form of integrity constraints. As well as OCL invariants, headless SWRL rules are integrity constraints. Therefore, it is natural to define a mapping between rules of the same type.

The solution to the rule interchange problem between OCL invariants and SWRL rules consists of two main parts:

- Building a mapping from rules, expressed using OCL syntax into rules, expressed using SWRL syntax.
- The proof of the semantic correctness, which helps to ensure, that rules in SWRL, obtained as a result of the mapping, have the same logical meaning as original OCL invariants.

The first issue is resolved by finding correspondence between syntactical structures of OCL and SWRL. This correspondence is initially defined using a common understanding and a general knowledge about semantics of both languages. However, in order to make sure that the mapping of rules is correct in the sense that it preserves the meaning of rules (or, speaking in terms of the model theory, rules in OCL and SWRL have the same models), the formal proof of the correctness is required. Apart from the necessity to prove mathematically the correctness of a mapping, defined heuristically or by an intuition, the proof is necessary for software engineers in order to be sure that their OCL invariants, being mapped into SWRL, produce an expected system behavior.

The interchange mapping from a source language into a target language is performed via a third language. This language plays the role of an intermediate representation of rules, which is known as a *rule interchange format*. When the amount of languages participating in the interchange grows, the approach with an interchange language is reusable and requires less transformations than in case of a direct one-to-one language interchange. Development of the rule interchange format is conducted by various scientific and industrial communities such as REWERSE[1] and W3C Working Group on Rule Interchange Format (RIF)[2].

In this work we use the REWERSE Rule Markup Language (R2ML) as a rule interchange format, which was developed in the European Project REWERSE. The work on R2ML has been performed in collaboration with the W3C RIF, which employs some R2ML experience, in particular a metamodeling approach to defining abstract syntax of the language. We introduce R2ML and explain its choice as an interchange language in Section 2.

3.2 The Formal Problem Statement

We identify two approaches to showing the correctness of a rule interchange mapping. The first approach is heuristic. Rules in the source language are executed and results are compared against the execution of rules in the target language. The main drawback of

[1]EU-project REWERSE: `http://rewerse.net`
[2]W3C RIF: `http://www.w3.org/2005/rules`

this approach is that it does not guarantee the correct execution result for all possible rules. However, this approach can be sufficient in some practical cases, for instance when formal semantics of rule languages is not known or defined. A test-driven validation of rule bases ([25], [72]) can be employed for generating execution results.

The second approach is formal and it needs a consideration of formal semantics of rule languages, which are defined for OCL and SWRL. In this work, we employ the formal approach since, in contrast with the heuristic approach, it shows the correctness of the mapping for all possible rules.

Let us define the problem of the semantic correctness of a rule interchange mapping for two rule languages L_1 and L_2:

Definition 5 (Semantic correctness of a rule mapping)**.** *Let $\mathcal{I}$ be an interpretation of L_1 and $\mathcal{J}$ be an interpretation of L_2, $\models$ be a satisfaction relation, and we consider the following mappings:*

- *t, which maps a rule, expressed in L_1 into a rule, expressed in L_2;*
- *T, which maps L_1 interpretation $\mathcal{I}$ into L_2 interpretation $\mathcal{J}$.*

We say that the rule mapping t is semantically correct with respect to T *iff for any rule r, expressed in L_1 the following holds:*

$$\mathcal{J} \models r \Longrightarrow T(\mathcal{J}) \models t(r)$$

In other words, we need to show that the rule mapping t preserves the satisfaction relation, or the set of models of any rule r expressed in L_1 is the same as the set of models of $t(r)$, which is an expression in L_2. The role and the definition of T is described in Section 3.3.2.

In this work, we solve this problem twice since we build two mappings: From OCL into R2ML and from R2ML into SWRL.

3.3 Mapping OCL into R2ML

In this section, we define a mapping from OCL into R2ML. The mapping process consists of the following steps: The syntax and the semantics of UML class models are defined in Section 3.3.1; the mapping of OCL class model into R2ML vocabulary is defined in Section 3.3.2; and finally, the mapping of OCL invariants into R2ML integrity rules and the correctness proof are presented in Section 3.3.3.

3.3.1 Syntax and Semantics of UML Class Models

A UML class model plays a role of a type system for OCL invariants. All concepts of a business domain are defined in the UML class model (also called a business vocabulary).

According to the OCL specification [4], a class model provides the context for OCL expressions and constraints, and consists of the following components:

- a set of classes;

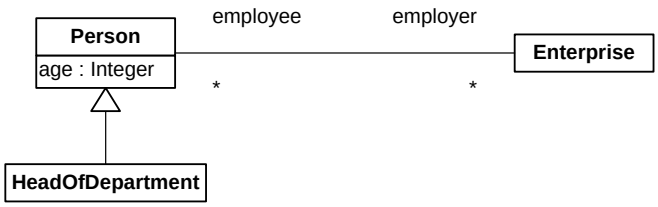

Figure 3.1: An example of a UML class model

- a set of attributes for each class;
- a set of operations for each class;
- a set of associations with role names and multiplicities;
- a set of object names;
- a set of data literals;
- a generalization hierarchy over classes.

Additionally, types such as *Integer* and *String* are available for describing types of attributes and operation parameters.

Syntax of UML Class Models

We define the syntax of UML class models and refer to Figure 3.1 in order to explain defined concepts by means of example. In the following definition we omit symbol sets with association names and user-defined operations. We can do this since associations are not needed for mapping OCL into SWRL: There is no concept of association in OWL/SWRL and navigation along associations in OCL is performed by means of role names. OWL/SWRL does not support user-defined operations as well, only built-in functions. Therefore, we do not consider the mapping of UML user-defined operations into OWL/SWRL.

A class model has the following structure:

$$\mathcal{M} = (\text{Class}, \text{Attr}, \text{Roles}, \text{Obj}_{\text{UML}}, \text{DLit}_{\text{UML}}, \prec)$$

where

1. Class is the set of classes; This set in Figure 3.1 is:

   Class={'Person', 'Enterprise', 'HeadOfDepartment'}

2. Attr is the set of operation signatures for mapping objects to attribute values. This set in Figure 3.1 is:

   Attr={'age(p:Person): integer', 'name(e:Enterprise): string'}

Table 3.1: Function $\mathcal{J}_{Roles}(employer)$

	p_1	p_2	p_2	p_3
employer	c_1	c_1	c_2	c_2

Table 3.2: Function $\mathcal{J}_{Attr_c}(a)$

	p_1	p_2	p_3
age	28	30	55
	c_1	c_2	
name	*Sony*	*Google*	

Let us consider a person p_1 of age 28, then we have $age(p_1) = 28$. It is important to point out that all attributes in UML/OCL are considered in the context of their classes, which is not the case of, for instance, OWL, where a data property can be defined without specifying a domain class.

3. Roles is the set of role names.

   The set of role names in Figure 3.1 consists of two names:

   Roles={'employee', 'employer'}

4. Obj_{UML} is the set of UML object names.

5. DLit_{UML} is the set of UML data literals.

6. $\prec$ is a partial order on Class reflecting the generalization hierarchy of classes.

Interpretation of UML Class Models

As in the case of R2ML semantics (Section 2.3), we denote the set of all objects as $\mathcal{O}$ and the set of all data values as $\mathcal{D}$. In order to give examples of interpretation functions let $\{p_1, p_2, p_3, c_1, c_2\} \in \mathcal{O}$ be objects.

Definition 6 (Interpretation of UML class models)**.** *An abstract interpretation of a class model is a tuple of functions*

$$\mathcal{J} = (\mathcal{J}_{Class}, \mathcal{J}_{Roles}, \mathcal{J}_{Attr}, \mathcal{J}_{Op}, \mathcal{J}_{\prec}, \mathcal{J}_{Obj}, \mathcal{J}_{DLit})$$

such that:

1. $\mathcal{J}_{Class}$ *maps each class c into the set of objects* $o_1, ..., o_n$ *of class c:*

   $$\mathcal{J}_{Class}(c) = \{o \in \mathcal{O} | type(o) \in \text{Class} \wedge type(o) \preceq c\}$$

   where type(o) returns a class of object o and o is a direct instance of c. For our example the interpretation function is defined as follows:

   $$\mathcal{J}_{Class}(\text{'Person'}) = \{p_1, p_2, p_3\}, \mathcal{J}_{Class}(\text{'HeadOfDepartment'}) = \{p_3\}$$

   $$\mathcal{J}_{Class}(\text{'Enterprise'}) = \{c_1, c_2\}$$

2. *$\mathcal{J}_{Roles}$ maps each role name $r \in$* Roles *into an object function $\mathcal{J}_{Roles}(r) \subseteq \mathcal{O} \longrightarrow \mathcal{O}$. In our example the object function for the role name "*employer*" is defined in Table 3.1.*

3. *$\mathcal{J}_{Attr}$ maps each attribute $a \in$* Attr *of a class into a datatype function $\mathcal{J}_{Attr}(a) \subseteq \mathcal{O} \longrightarrow \mathcal{D}$. In our example datatype functions for attributes "age" and "name" are defined in Table 3.2. We assume UML class diagrams at the Platform Independent Model (PIM) level, where class attributes are considered to be datatype functions.*

4. *$\mathcal{J}_{Obj}$ maps each object name into an object.*

5. *$\mathcal{J}_{DLit}$ maps each data literal into a data value.*

3.3.2 Mapping OCL Models into R2ML

In the previous section we have defined the UML vocabulary $\mathcal{M}$. When performing an interchange of rules it is important to know how elements of the target vocabulary (R2ML) are obtained from the elements of the source vocabulary (UML) or, in other words, which components of R2ML vocabulary V*oc* correspond to which components of $\mathcal{M}$. For instance, if "Person" is a UML class, then its R2ML counterpart must be a unary predicate "Person", not an attribute or an association.

Obtaining R2ML Vocabulary from UML Vocabulary

In this section we explain how an R2ML vocabulary can be obtained from the UML vocabulary, i.e. how the symbol set $\mathcal{M}$ of UML vocabulary is mapped into V*oc* of R2ML vocabulary.

We define V*oc* such that

1. $\text{OPred}^{(1)} = \text{Class}$, i.e. the set of R2ML unary object predicates is equal to the set of UML classes;

2. OProp = Roles, i.e. the set of R2ML object properties is equal to the set of UML roles;

3. DProp = Attr, i.e. the set of R2ML datatype properties is equal to the set of UML attributes;

4. $\text{Obj} = \text{Obj}_{\text{UML}}$, i.e. the set of R2ML object names is equal to the set of UML object names;

5. $\text{DLit} = \text{DLit}_{\text{UML}}$, i.e. the set of R2ML data literals is equal to the set of UML data literals;

In practice, for mapping a UML vocabulary into an R2ML vocabulary some transformation t may be needed: for instance, in order to comply with naming conventions of UML with its package structure and SWRL, where concepts are denoted by URIs. For

example, a UML class name `de.tu-cottbus.inf.Person` can be translated into the URI: `http://inf.tu-cottbus.de#Person`. The transformation t then would work as following:

$$t(\text{'de.tu-cottbus.inf.Person'}) = \text{'http://inf.tu-cottbus.de\#Person'}$$

i.e. t transforms the UML class name into the R2ML class name, but for simplicity we omit the definition of t and assume that UML classes and corresponding R2ML unary object predicates (as well as other concepts) are named equally.

Mapping the Interpretation of UML Vocabulary into the Interpretation of R2ML Vocabulary

In order to make sure that a rule interchange mapping preserves the satisfaction relation, the mapping of interpretations has to be defined. For instance, if $\mathcal{J}_{Class}(c)$ is the set of instances of a UML class c, then it has to be the same as $\mathcal{I}_{Class}(c)$, which is the set of instances of an R2ML class c. In other words, the same classes in UML and R2ML consist of the same objects. In order to obtain this, we define the mapping T, which maps the interpretation $\mathcal{J}$ of UML into the interpretation $\mathcal{I}$ of R2ML.

Definition 7. *The mapping T from the interpretation $\mathcal{J}$ into the interpretation $\mathcal{I}$ is defined as following:*

1. $T(\mathcal{J}_{Class}) = \mathcal{I}_{OPred^{(1)}}$ *such that for all* $c \in Class$, $\mathcal{I}_{OPred^{(1)}}(c) = \mathcal{J}_{Class}(c)$, *where* $OPred^{(1)}$ *is a set of R2ML unary object predicates;*
2. $T(\mathcal{J}_{Roles}) = \mathcal{I}_{OProp}$ *such that for all* $r \in Roles$, $\mathcal{I}_{OProp}(r) = \mathcal{J}_{Roles}(r)$;
3. $T(\mathcal{J}_{Attr}) = \mathcal{I}_{DProp}$ *such that for all* $attr \in Attr$, $\mathcal{I}_{DProp}(attr) = \mathcal{J}_{Attr}(attr)$;
4. $T(\mathcal{J}_{Obj}) = \mathcal{I}_{Obj}$ *such that for all* $o_{name} \in Obj_{UML}$, $\mathcal{I}_{Obj}(o_{name}) = \mathcal{J}_{Obj}(o_{name})$;
5. $T(\mathcal{J}_{DLit}) = \mathcal{I}_{DLit}$ *such that for all* $lit \in DLit$, $\mathcal{I}_{DLit}(lit) = \mathcal{J}_{DLit}(lit)$;

Mapping Basic UML Atoms into R2ML

The definition of T needs a justification and it can be given by showing that it preserves the satisfaction of atoms. A class diagram may contain such atomic logical expressions as "John is a Person" or "The age of John is 33". We may distinguish between three types of such expressions, which we call *basic atoms*: A classification atom, a role atom, and an attribution atom. A sample diagram, depicted in Figure 3.2, represents a classification atom ("John is a Person"), an attribution atom ("The age of John is 33") and a role atom ("John has BTU as employer"). Note, that "basic atom" is just a denotation of a well-known UML concepts (UML object with attribute values and UML link) in logical terms.

We express these atoms, using OCL syntax and define their mapping into R2ML. In the next section we extend the syntax to more complex OCL expressions (navigation expressions, formulas, invariants) and use these atoms as an induction base in the correctness proof.

Basic UML atoms are:

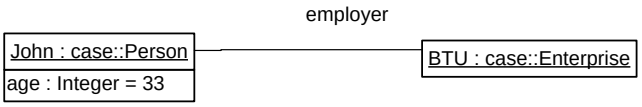

Figure 3.2: UML object model, depicting basic atoms

- A *classification atom*, which corresponds to the OperationCallExp with a Boolean operation "o.oclIsKindOf(C: OclType):Boolean" in the OCL metamodel ([4] Section 8, Abstract Syntax). The syntax is o.oclIsKindOf(C), where o is a variable name, bound to an object and C is a class.

- A *role atom*, which corresponds to the boolean equality operation and AssociationEndCallExp in the OCL metamodel. The syntax is $o.r{=}v$, where o and v are object names bound to objects and r is a role name.

- An *attribution atom*, which corresponds to the boolean equality operation and AttributeCallExp in the OCL metamodel. The syntax is $o.attr{=}d$, where o is an object variable name bound to an object, $attr$ is an attribute name and d is a data variable name.

Mapping t for the basic atoms:

1. Let o.oclIsKindOf(C) be an OCL expression, then the result is an R2ML object classification atom: $t(o.\text{oclIsKindOf}(C)) = \text{ObjClAt}(\text{class}(C), \text{OVar}(o))$

2. Let $o.r = v$ be an OCL expression, then the result is an R2ML reference property atom: $t(o.r = v) = \text{RefPropAt}(r, \text{subj}(\text{OVar}(o)), \text{obj}(\text{OVar}(v)))$

3. Let $o.attr = d$ be an OCL expression, then the result is an R2ML attribution atom: $t(o.attr = d) = \text{AttrAt}(\text{attr}, \text{subj}(\text{OVar}(o)), \text{DVar}(d))$

We consider t as an evident mapping, which does not need a proof or justification. These atoms are simplest logical expressions and the definition of t comes from their natural understanding and the common sense. In Section 3.3.3 we extend the definition of t to complex OCL expressions such as existentially quantified formulas and others. In order to show that this extended mapping preserves the satisfaction relation, we employ the induction method. The mapping of basic atoms, defined above, is taken as an induction base.

The forthcoming Theorem 1 states that interpretation T preserves the satisfaction of three basic atoms.

Faithfulness of T

Theorem 1 (Faithfulness of the interpretation mapping).
Let t be the mapping for atoms and T is the mapping for interpretations as defined above. T is faithful in the sense that it preserves the satisfaction of atoms, i.e. the following axioms hold:

Classification atom

$$\forall c \in \text{Class and variable name } x\ \mathcal{J} \models x.\text{oclIsKindOf}(c) \Longrightarrow T(\mathcal{J}) \models t(x.\text{oclIsKindOf}(c)) \tag{3.1}$$

Role atom

$$\forall r \in \text{Roles and variable names } x, y,\ \mathcal{J} \models x.r = y \Longrightarrow T(\mathcal{J}) \models t(x.r = y) \tag{3.2}$$

Attribution atom

$$\forall attr \in \text{Attr and variable names } x, d,\ \mathcal{J} \models x.attr = d \Longrightarrow T(\mathcal{J}) \models t(x.attr = d) \tag{3.3}$$

Proof. The right-hand side of the implication 3.1 is equivalent to $\mathcal{I} \models \text{ObjClAt}(\text{class}(c), \text{obj}(\text{OVar}(x)))$ and is true iff

$$\mathcal{I}_{Obj}(x) \in \mathcal{I}_{OPred^{(1)}}(c) \Longleftrightarrow$$

$$\mathcal{J}_{Obj}(x) \in \mathcal{J}_{Class}(c)$$

which is implied by

$$\mathcal{J} \models x.\text{oclIsKindOf}(c)$$

The right-hand side of the implication 3.2 is equivalent to $\mathcal{I} \models \text{RefPropAt}(r, \text{subj}(\text{OVar}(x)), \text{obj}(\text{OVar}(y)))$ and is true iff

$$\langle \mathcal{I}_{Obj}(x), \mathcal{I}_{Obj}(y) \rangle \in \mathcal{I}_{OProp}(r) \Longleftrightarrow$$

$$\langle \mathcal{J}_{Obj}(x), \mathcal{J}_{Obj}(y) \rangle \in \mathcal{J}_{Roles}(r)$$

which is implied by

$$\mathcal{J} \models x.r = y$$

The right-hand side of the implication 3.3 is equivalent to $\mathcal{I} \models \text{AttrAt}(attr, \text{subj}(\text{OVar}(x)), \text{DVar}(d))$ and is true iff

$$\langle \mathcal{I}_{Obj}(x), \mathcal{I}_{DLit}(d) \rangle \in \mathcal{I}_{DProp}(attr) \Longleftrightarrow$$

$$\langle \mathcal{J}_{Obj}(x), \mathcal{J}_{DLit}(d) \rangle \in \mathcal{J}_{Attr}(attr)$$

which is implied by

$$\mathcal{J} \models x.attr = d$$

□

3.3.3 Mapping OCL-Lite Invariants into R2ML Integrity Rules

Before defining the mapping from OCL invariants into R2ML integrity rules, we define the syntax and the semantics of OCL expressions. We map only a subset of OCL, which we call OCL-Lite [48]. It is expressive enough to represent SWRL rules and does not contain expressions, which cannot be mapped into SWRL.

Inductive Definition of the OCL-Lite Expressions Syntax

The syntax of OCL-Lite expressions is defined bottom-up, so that more complex expressions are built from simple ones. We denote an OCL-Lite expression, which allows following association links between objects and retrieve connected objects, as a *navigation expression*, which is defined inductively:

- An object variable x is a navigation expression;
- An expression of the form $e.r$ is a navigation expression, where r is a role name and e is a navigation expression.

An example of a navigation expression is `self.car.model`, where `self` denotes an object variable of the context classifier. Note that a navigation expression is not a new concept on top of OCL, but a denotation of the particular case of an OCL expression with a navigation operation.

1. **self** is a special variable, which type is given by the invariant context;
2. For every model type t there is an unlimited number of variables v_t, which are OCL expressions of type t;
3. If f is an operation symbol with argument types $t_1, ..., t_n$ and result type t_r, $e_1, ..., e_n$ are OCL expressions and type of e_i is t_i for all $1 \leq i \leq n$ then $f(e_1, ..., e_n)$ is OCL expression of type t_r. The set of operation symbols includes:
   - some of predefined data operations: $+, -, *$;
   - attribute operations, for instance, `self.age`, `e.salary`;
   - navigation by role names, for instance, `self.pickupBranch`.

Boolean OCL expressions

1. If e_1 and e_2 are OCL expressions of an appropriate type (but not Boolean) then $e_1 > e_2$, $e_1 < e_2$, $e_1 >= e_2$, and $e_1 <= e_2$ are Boolean OCL expressions. Note, that symbols $>$, $<$, $>=$, and $<=$ are OCL operations in the OCL metamodel, however here we define OCL expressions with these symbols separately since logically they are predicate symbols, while expressions, defined in the previous item are different types of terms;
2. If e_1, e_2 are OCL expressions of the same type, then $e_1 = e_2$, $e_1 <> e_2$ are Boolean expressions.
3. If e is a navigation expression, then e.oclIsKindOf(C) is a Boolean expression;
4. If $e.r$ is a navigation expression, for instance, `x.employee`, then $e.r \rightarrow$ size() *op* n is Boolean expression, where *op* is either $>$, $<$, $=$, $>=$, $<=$, or $<>$. We do not define an expression of the form $e \rightarrow$ size(), where e is of any OCL collection type, for instance, resulted from some operation on collections like includeAll(). The defined expression is mapped into R2ML logical formulas, which express cardinality restrictions. For instance, self.employer $\rightarrow$ size() > 1 can be represented by means of R2ML at least quantified formulas.

5. If $e.r$ is a navigation expression, for instance, `self.employee`,
   then $e.r \rightarrow \text{exists}(x|f(x))$ is a Boolean expression, where $f(x)$ is a Boolean OCL expression from variable x, which is bound by the quantifier.

6. If $e.r$ is a navigation expression, then $e.r \rightarrow \text{forAll}(x|f(x))$ is a Boolean expression, where $f(x)$ is a Boolean OCL expression from variable x, which is bound by the quantifier.

7. If $e_1, ...e_n$ are Boolean expressions then e_1 and... and e_n is a Boolean expression;

The Syntax of OCL-Lite Invariants

In this paper, we consider the mapping of OCL-Lite invariants only. Invariants, also called *integrity rules* or constraints, have the following syntax:

context *typeName*
  inv *OCLLiteExpression*

Here the *typeName* is the name of the context class. Let us denote an OCL invariant as $Inv(C, exp(x))$. According to the OCL metamodel [4], OCL expressions are of different types. In the header of the OCL invariant the *OCLLiteExpression* has to be of type Boolean.

Interpretation of OCL-Lite Expressions

In this section we define the satisfaction relation for OCL-Lite expressions, defined in section 3.3.3. We use the definition of the valuation function (see Definition 3, R2ML semantics). The satisfaction relation $\models$ is such that

1. If $\text{Inv}(C, exp(x))$ is an OCL-Lite invariant where C is the context classifier, $exp(x)$ is a Boolean expression with the variable x of C, then $\mathcal{J}_\mathcal{V} \models \text{Inv}(C, exp(x))$ iff for all $a \in \mathcal{J}(C)$, $\mathcal{J}_{\mathcal{V}[x/a]} \models^t exp(x)$

2. If e_1, e_2 are Boolean expressions then

   (a) $\mathcal{J}_\mathcal{V} \models e_1$ *and* e_2 iff $\mathcal{J}_\mathcal{V} \models e_1$ and $\mathcal{J}_\mathcal{V} \models e_2$

3. If e_1, e_2 are OCL-Lite expressions of the same type then

   (a) $\mathcal{J}_\mathcal{V} \models e_1 = e_2$ iff $\mathcal{J}_\mathcal{V}(e_1) = \mathcal{J}_\mathcal{V}(e_2)$

4. An OCL-Lite operation expression f is interpreted by the function, associated with the operation symbol and is defined as follows:

   $$\mathcal{J},\mathcal{V}\,(f(e_1, ..., e_n)) = \mathcal{J},\mathcal{V}\,(f)(\mathcal{J},\mathcal{V}\,(e_1), ..., \mathcal{J},\mathcal{V}\,(e_n))$$

   If the result type t_r of the function f is of boolean type (f is one of $>$, $<$, etc) then f is interpreted as corresponding datatype predicates and each argument is evaluated separately. All other result types are interpreted as terms with corresponding function: Predefined data operations, attribute operations, and navigation by role name operations.

5. If $e.r \rightarrow \text{exists}(y|f(y))$ is a Boolean expression where $e.r$ is a navigation expression and $f(y)$ is a Boolean expression with variable y, which runs over the elements of the collection, denoted by r, then

   (a) $\mathcal{J}_{\mathcal{V}} \models e.r \rightarrow \text{exists}(y|f(y))$ iff exists $a \in \mathcal{O}$ such that $\mathcal{J}_{\mathcal{V}[y/a]} \models (y \in e.r)$ and $\mathcal{J}_{\mathcal{V}[y/a]} \models f(y)$;

6. If $e.r \rightarrow \text{forAll}(y|f(y))$ is a Boolean expression where $e.r$ is a navigation expression and $f(y)$ is a Boolean expression with variable y, which runs over the elements of the collection, denoted by r, then

   (a) $\mathcal{J}_{\mathcal{V}} \models e.r \rightarrow \text{forAll}(y|f(y))$ iff for all $a \in \mathcal{O}$ $\mathcal{J}_{\mathcal{V}[y/a]} \models (y \in e.r)$ implies $\mathcal{J}_{\mathcal{V}[y/a]} \models f(y)$;

7. If $e.r \rightarrow \text{size}() < n$ is a Boolean expression where $e.r$ is a navigation expression and y runs over the elements of the collection, denoted by r, then

   (a) $\mathcal{J}_{\mathcal{V}} \models \exists^{[0..n]} y|e.r \rightarrow \text{size}() < n$ iff exists $a_1, ..., a_m \subseteq \mathcal{O}, m < n$ such that for all $i = 1...m$ $\mathcal{J}_{\mathcal{V}[y/a_i]} \models e.r \rightarrow \text{size}() < n$;

8. If v is a variable then $\mathcal{J}(v) = \mathcal{V}(v)$

A Compositional Mapping from OCL-Lite into R2ML

Below we define a mapping from OCL-Lite invariants into R2ML integrity rules and prove its correctness.

The mapping principles in brief: OCL-Lite expressions of Integer, Real and other primitive data types are mapped into R2ML terms; OCL-Lite navigation expressions are mapped into R2ML complex terms such as reference property function terms; OCL-Lite expressions of Boolean type are mapped into R2ML atoms. First we map OCL-Lite expressions of the Boolean type:

1. **OCL comparison operations**. If $f(e_1, ..., e_n)$ denotes $<$, $>$, $<=$, $=>$, then $t(f)$ corresponds to the R2ML datatype predicate atom with corresponding datatype predicate. For instance, if e_1 and e_2 are integer expressions, then

   $$t(e_1 > e_2) := \text{DPredAt(swrlb:greaterThan, dataArgs}(t(e_1), t(e_2)))$$

2. **OCL Boolean equality expression**. If $e_1 = e_2$ is a Boolean expression, then $t(e_1 = e_2)$ depends on the type of e_1 and e_2:

   (a) If $e_1 := e.r$ is a navigation expression, where r is a role, then $t(e.r = e_2)$ is an R2ML reference property atom

   $$\text{RefPropAt}(r, \text{subj}(t(e)), \text{obj}(t(e_2)))$$

   (b) If $e_1 := e.attr$ is an operation expression, where $attr$ is an attribute, then $t(e.attr = e_2)$ is an R2ML attribution atom

   $$\text{AttrAt}(attr, \text{subj}(t(e)), \text{dataVal}(t(e_2)))$$

3. **OCL Boolean expression oclIsKindOf()**. If e.oclIsKindOf(C) is a Boolean expression, where e is a navigation expression, and C is a class name, then $t(e.\text{oclIsKindOf}(C))$ is an R2ML object classification atom

$$\text{ObjClAt}(\text{class}(C),\ \text{term}(t(e)))$$

4. **OCL cardinality expression**. If $e.r$ is a navigation expression and $e.r \rightarrow \text{size}() < n$, then it is mapped into R2ML at most quantified formula with an R2ML reference property atom, which comes from the navigation expression $e.r$:

$$\begin{aligned} t(e.r \rightarrow \text{size}() < n) \quad := \quad & \text{AtMostQuanF}(\text{vars}(\text{OVar}(x)), \\ & \text{RefPropAt}(r,\ \text{subj}(t(e)),\ \text{obj}(x)),\ \text{maxCard}(n+1)) \end{aligned}$$

The symbol x explicitly denotes a variable, which is restricted to the instances of a collection r by the quantifier.

5. **OCL Boolean expression exists()**. If an expression is $e.r \rightarrow \text{exists}(x|f(x))$, where $e.r$ is a navigation expression and $f(x)$ is a Boolean expression with variable x, which runs over elements of the role collection, denoted by r, then it is mapped into R2ML existentially quantified formula with a conjunction of atoms:

$$\begin{aligned} t(e.r \rightarrow \text{exists}(x|f(x)) \quad := \quad & \text{ExQuantF}(\text{vars}(\text{OVar}(x)), \\ & \text{And}(\text{RefPropAt}(r,\ \text{subj}(t(e)),\ \text{obj}(x)),\ t(f(x)))) \end{aligned}$$

For example, let us consider an OCL expression
$exp := x.\text{employee}\rightarrow\text{exists}(y|y.\text{age}= 18)$

$$\begin{aligned} t(exp) \quad := \quad & \text{ExQuantF}(\text{vars}(\text{OVar}(y)), \\ & \quad \text{And}(\text{RefPropAt}(\text{employee},\ \text{subj}(x),\ \text{obj}(y)),\ t(y.\text{age}= 18)) := \\ & \text{ExQuantF}(\text{vars}(\text{OVar}(y)), \\ & \quad \text{And}(\text{RefPropAt}(\text{employee},\ \text{subj}(x),\ \text{obj}(y)), \\ & \qquad \text{AttrAt}(\text{age},\ \text{cxtArg}(y),\ \text{DLit}(18)))) \end{aligned}$$

6. **OCL Boolean expression forAll()**. If an expression is $e.r \rightarrow \text{forAll}(x|f(x))$, where $e.r$ is a navigation expression and $f(x)$ is a Boolean expression, then

$$\begin{aligned} t(e.r \rightarrow \text{forAll}(x \mid f(x))) \quad := \quad & \text{UnivQuantF}(\text{vars}(\text{OVar}(x)), \\ & \quad \text{Impl}(\text{antec}(\text{RefPropAt}(r,\ \text{subj}(t(e)),\ \text{obj}(x))), \\ & \qquad \text{cons}(t(f(x))))) \end{aligned}$$

7. **OCL conjunction**. If e_1 and e_2 is a Boolean expression, then it is mapped into a conjunction of R2ML atoms:

$$t(e_1 \text{ and } e_2) := \text{And}(t(e_1), t(e_2))$$

8. **OCL invariants**. If $F(C, exp(x))$ is an OCL invariant with C as a context classifier, $exp(x)$ as a Boolean expression, and x as a universally quantified variable of class C (x is usually denoted as `self`), then it is mapped into an R2ML alethic integrity rule with a universally quantified formula and a constraint with a logical formula $t(exp(x))$.

$$\begin{aligned} t(Inv(C, exp(x))) \quad := \quad & \text{AIR(constr(UnivQuantF(vars(OVar}(x)), \\ & \quad \text{Impl(cons(ObjClAt(class(C), term}(x))), \\ & \quad\quad \text{antec}(t(exp(x))))))) \end{aligned}$$

OCL terms mappipng:

9. If v is an object variable of type C then $t(v) = \text{OVar}(v, \text{class}(C))$
10. If v is an data variable of type d then $t(v) = \text{DVar}(v, \text{dType}(d))$
11. If $f(e_1, ..., e_n)$ is an operation OCL expression, then $t(f)$ is defined depending on the operation, denoted by the function symbol f.

    (a) If f denotes a data operation: $+$, $-$, $*$, $\backslash$, then $t(f)$ corresponds to the R2ML data operation term with corresponding datatype operation. For instance, let e_1, e_2 are OCL expressions of type Integer, then

    $$\begin{aligned} t(e_1 + e_2) \quad := \quad & \text{DatatypeFuncTerm(op:numeric-add,} \\ & \quad \text{dataArgs}(t(e_1), t(e_2))) \end{aligned}$$

    (b) If $f := e.r$ be a navigation expression, where r is a role name and e is a navigation expression, then $t(f)$ corresponds to the R2ML reference property function term

    $$t(f) := \text{RefPropFuncTerm}(r, \text{cxtArg}(t(e)))$$

    (c) If $f := e.attr$ denotes an attribute operation, where e is a navigation expression and $attr$ is an attribute, then $t(f)$ corresponds to the R2ML attribute function term

    $$t(f) := \text{AttrFuncTerm(attr, cxtArg}(t(e)))$$

The Correctness Proof

Theorem 2 (Semantical correctness of OCL to R2ML mapping).
The mapping t of OCL invariants into R2ML integrity rules is *semantically correct*, i.e. it preserves the satisfaction of rules (see Definition 5).

Proof. In Section 3.3.2 we have defined the mapping of the basic UML atoms into R2ML and assumed that this mapping is evident and, therefore, needs no proof. We have also defined the mapping T from OCL interpretations into R2ML interpretations and proved that it is faithful in the sense it preserves the satisfaction of basic atoms. In order to prove that the mapping t preserves the satisfaction of rules, we have to prove that it preserves the satisfaction of rule constituents: Logical formulas. In the proofs below we employ the induction method by the structure of OCL boolean expressions, defined in Section 3.3.3. The induction base is the mapping of three basic atoms, which is defined in Section 3.3.2.

1. **OCL comparison operations**
   Let us prove the correctness of the mapping for OCL operation ">":

   $$\mathcal{J}_\mathcal{V} \models e_1 > e_2 \Longrightarrow T(\mathcal{J}_\mathcal{V}) \models t(e_1 > e_2)$$

   This implication is equivalent to

   $$\mathcal{J}_\mathcal{V} \models e_1 > e_2 \Longrightarrow \mathcal{I}_\mathcal{V} \models \text{DPredAt(swlrb:greaterThan, dataArgs}(t(e_1), t(e_2))$$

   where $\mathcal{I}_\mathcal{V} = T(\mathcal{J}_\mathcal{V})$.
   The left hand side of the implication is equivalent to

   $$\langle \mathcal{J}_\mathcal{V}(e_1), \mathcal{J}_\mathcal{V}(e_2) \rangle \in \mathcal{J}(\text{`} >')$$

   and the right-hand side of the implication is equivalent to

   $$\langle \mathcal{I}_\mathcal{V}(t(e_1)), \mathcal{I}_\mathcal{V}(t(e_2)) \rangle \in \mathcal{I}(\text{`swrlb:greaterThan'})$$

   Therefore, the original implication holds since the extension tables of ">" and "swrlb:greaterThan" are the same.

2. **OCL Boolean equality expression ($e_1 = e_2$)**
   Let $p_1...p_k$ be an OCL navigation expression and $p_1...p_k = u$ is an OCL Boolean expression, where u is either an object variable or an object name. We build the induction by the length of the navigation expression. We have to prove

   $$\mathcal{J}_\mathcal{V} \models^t (p_1...p_k = u) \Longrightarrow \mathcal{I}_\mathcal{V} \models^t t(p_1...p_k = u) \tag{3.4}$$

   The induction hypothesis is:

   $$\mathcal{J}_\mathcal{V} \models (p_1...p_{k-1} = v) \Longrightarrow \mathcal{I}_\mathcal{V} \models t(p_1...p_{k-1} = v)$$

   where v is either an object variable or an object name. It is equivalent to (after applying the mapping t for the right-hand side):

   $$\mathcal{J}_\mathcal{V} \models (p_1...p_{k-1} = v) \Longrightarrow$$

   $$\mathcal{I}_\mathcal{V} \models \text{RefPropAt(id}(p_{k-1}), \text{subj}(t(p_1...p_{k-2})), \text{obj}(t(v)))$$

The implication 3.4 is equivalent to:

$$\mathcal{J}_{\mathcal{V}} \models ((v.p_k = u)\ and\ (p_1...p_{k-1} = v)) \Longrightarrow \mathcal{I}_{\mathcal{V}} \models t((v.p_k = u)\ and\ (p_1...p_{k-1} = v))$$

Which is, in turn, equivalent to (after applying the mapping t to the right-hand side):

$$\mathcal{J}_{\mathcal{V}} \models ((v.p_k = u)\ and\ (p_1...p_{k-1} = v)) \Longrightarrow$$

$$\mathcal{I}_{\mathcal{V}} \models \text{And}(\text{RefPropAt}(\text{id}(p_k), \text{subj}(t(v)), \text{obj}(t(u))), \text{RefPropAt}(\text{id}(p_{k-1}), \text{subj}(t(p_1...p_{k-2})), \text{obj}(t(v))))$$

After applying the definition of the conjunction for left-hand and right-hand sides of this implication (a conjunction holds when every conjunct holds):

$$\mathcal{J}_{\mathcal{V}} \models v.p_k = u \Longrightarrow \mathcal{I}_{\mathcal{V}} \models \text{RefPropAt}(\text{id}(p_k), \text{subj}(t(v)), \text{obj}(t(u)))$$

$$\mathcal{J}_{\mathcal{V}} \models p_1...p_{k-1} = v \Longrightarrow \mathcal{I}_{\mathcal{V}} \models \text{RefPropAt}(\text{id}(p_{k-1}), \text{subj}(t(p_1...p_{k-2})), \text{obj}(t(v)))$$

Having $T(\mathcal{J}_{\mathcal{V}}) = \mathcal{I}_{\mathcal{V}}$, these two implications hold: First by the induction base (basic atom) and second by the induction hypothesis.

3. **OCL Boolean expression oclIsKindOf()**
   The proof for the OCL Boolean expression e.oclIsKindOf(C), where e is a navigation expression, evaluated into a single object, is similar to equalities proof above.

4. **OCL cardinality expression** $e.r \rightarrow size() < n$
   We have to prove

$$\mathcal{J}_{\mathcal{V}} \models e.r \rightarrow \text{size}() < n \Longrightarrow T(\mathcal{J}_{\mathcal{V}}) \models t(e.r \rightarrow \text{size}() < n) \tag{3.5}$$

   where e denotes a navigation expression. An explicit variable y runs over elements of the collection, denoted by r.

   The implication 3.5 is equivalent to

$$\mathcal{J}_{\mathcal{V}} \models e.r \rightarrow \text{size}() < n \Longrightarrow$$

$$\mathcal{I}_{\mathcal{V}} \models \text{AtMostQuanF}(\text{vars}(\text{OVar}(y)), \text{maxCard}(n), \text{RefPropAt}(r, \text{subj}(t(e)), \text{obj}(y)))$$

   where $\mathcal{I}_{\mathcal{V}} = T(\mathcal{J}_{\mathcal{V}})$. This, in turn, is equivalent to (by the semantics definition for the OCL expression and the R2ML at most quantified formula): exists $a_1, ..., a_m \subseteq \mathcal{O}, m < n$ such that

   $\mathcal{J}_{\mathcal{V}[y/a_i]} \models e.r \rightarrow \text{size}() < n$ and for all $a_j \neq a_i$ $\mathcal{J}_{\mathcal{V}[y/a_j]} \not\models e.r \rightarrow \text{size}() < n$;

$$\mathcal{J}_{\mathcal{V}[y/a]} \models (e.r = y\ and\ f(y)) \Longrightarrow$$
$$\mathcal{I}_{\mathcal{V}[y/a]} \models \text{And}(\text{RefPropAt}(r, \text{subj}(t(e)), \text{obj}(y)), t(f(y)))$$

   This implication holds since it contains conjunction of logical expressions.

5. **OCL Boolean expression exists()**
We have to prove

$$\mathcal{J}_{\mathcal{V}} \models e.r \rightarrow \text{exists}(y|f(y)) \Longrightarrow T(\mathcal{J}_{\mathcal{V}}) \models t(e.r \rightarrow \text{exists}(y|f(y))) \tag{3.6}$$

where e denotes a navigation expression and $v(y)$ denotes a Boolean expression with variable y, which runs over elements of the collection, denoted by r.

The implication 3.6 is equivalent to

$$\mathcal{J}_{\mathcal{V}} \models e.r \rightarrow \text{exists}(y|f(y)) \Longrightarrow$$

$$\mathcal{I}_{\mathcal{V}} \models \text{ExQuantF}(\text{vars}(y), \text{And}(\text{RefPropAt}(r, \text{subj}(t(e)), \text{obj}(y)), t(f(y))))$$

where $\mathcal{I}_{\mathcal{V}} = T(\mathcal{J}_{\mathcal{V}})$. This, in turn, is equivalent to (by the semantics definition for the OCL exists() operation and the R2ML existentially quantified formula): exists $a \in \mathcal{O}$ such that

$$\mathcal{J}_{\mathcal{V}[y/a]} \models (e.r = y \; and \; f(y)) \Longrightarrow$$

$$\mathcal{I}_{\mathcal{V}[y/a]} \models \text{And}(\text{RefPropAt}(r, \text{subj}(t(e)), \text{obj}(y)), t(f(y)))$$

This implication holds since each expression of the conjunction holds.

6. **OCL Boolean expression forAll()**
We have to prove

$$\mathcal{J}_{\mathcal{V}} \models e.r \rightarrow \text{forAll}(y|f(y)) \Longrightarrow T(\mathcal{J}_{\mathcal{V}}) \models t(e.r \rightarrow \text{forAll}(y|f(y))) \tag{3.7}$$

where e denotes a navigation expression and $v(y)$ denotes a Boolean expression with variable y, which runs over elements of the collection, denoted by r.

The implication 3.7 is equivalent to

$$\mathcal{J}_{\mathcal{V}} \models e.r \rightarrow \text{forAll}(y|f(y)) \Longrightarrow$$

$$\mathcal{I}_{\mathcal{V}} \models \text{UnivQuantF}(\text{vars}(y), \text{And}(\text{RefPropAt}(r, \text{subj}(t(e)), \text{obj}(y)), t(f(y))))$$

where $\mathcal{I}_{\mathcal{V}} = T(\mathcal{J}_{\mathcal{V}})$.

This, in turn, is equivalent to (by the semantics definition for the OCL forAll() operation and the R2ML universally quantified formula): for all $a \in \mathcal{O}$ such that $\mathcal{J}_{\mathcal{V}[y/a]} \models (e.r = y)$ implies $\mathcal{J}_{\mathcal{V}[y/a]} \models f(y) \Longrightarrow$
$\mathcal{I}_{\mathcal{V}[y/a]} \models \text{RefPropAt}(r, \text{subj}(t(e)), \text{obj}(y))$ implies $\mathcal{I}_{\mathcal{V}[y/a]} \models t(f(y)))$.

This implication holds since its constituents formulae holds.

7. **Conjunction of OCL Boolean expressions**
Let us consider the semantic correctness of mapping of a **conjunction of arbitrary number of boolean expressions**. We use the induction method by the length of the conjunction and assume that the implication holds for formulas $f_1, ..., f_n$, so that for all $i = 1...n$ $\mathcal{J}_{\mathcal{V}} \models f_i \Longrightarrow T(\mathcal{J}_{\mathcal{V}}) \models t(f_i)$.

We have to prove that

$$\mathcal{J}_\mathcal{V} \models f_1 \; and \; ... \; and \; f_n \Longrightarrow T(\mathcal{J}_\mathcal{V}) \models t(f_1 \; and \; ... \; and \; f_n)$$

This is equivalent to

$$\mathcal{J}_\mathcal{V} \models f_1 \; and \; ... \; and \; f_n \Longrightarrow \mathcal{I}_\mathcal{V} \models \mathrm{And}(t(f_1), ..., t(f_n))$$

where $\mathcal{I}_\mathcal{V} = T(\mathcal{J}_\mathcal{V})$.
According to the definition of the satisfaction relation for R2ML conjunction,

$$\mathcal{I}_\mathcal{V} \models \mathrm{And}(t(f_1), ..., t(f_n))$$

iff for all $i = 1...n$

$$\mathcal{I}_\mathcal{V} \models t(f_i)$$

On the other hand, the left-hand side of the implication holds iff for all $i = 1...n$

$$\mathcal{J}_\mathcal{V} \models f_i$$

Consequently, the implication holds by induction hypothesis: for all $i = 1...n$,

$$\mathcal{J}_\mathcal{V} \models f_i \Longrightarrow T(\mathcal{J}_\mathcal{V}) \models t(f_i)$$

□

3.4 Mapping R2ML into SWRL

3.4.1 Syntax and Semantics of SWRL

SWRL abstract syntax

We give a simplified version of the original SWRL abstract syntax, defined in [8]. Simplifications are related to such unimportant for rule interchange features as annotations. A SWRL rule consists of an antecedent (body) and a consequent (head), each of which consists of a (possibly empty) set of atoms.

$$rule := \mathbf{Implies}(Antec(\{Atom\}), Conseq(\{Atom\}))$$

There are the following atoms:

- A classification atom classAtom(C, t), where C is an OWL DL description, as defined in [5], Section 2.3.2, and t is either an object variable or an individual ID;
- An individual-valued property atom indPropAtom$(prop, t_1, t_2)$, where $prop$ is an individual-valued property ID, and t_1 and t_2 are either object variables or individual IDs;

- A datavalued property atom datavalPropAtom($prop, t, d$), where *prop* is a datavalued property ID, t is either an object variable or an individual ID, and d is either a data variable or a data value;
- An equality atom sameAs(t_1, t_2), where t_1 and t_2 are either object variables or individual IDs;
- An inequality atom differentFrom(t_1, t_2), where t_1 and t_2 are either object variables or individual IDs;
- A built-in atom builtin(builtinID, $\{d\}$), where builtinID is an URI reference of a built-in, as defined in [8], Section "SWRL Built-ins" and $\{d\}$ is a list of either data variables or data values as built-in parameters.

Direct Model-theoretic Semantics of SWRL

According to the SWRL proposal by W3C [8] "a rule is satisfied by an interpretation iff every binding that satisfies the antecedent also satisfies the consequent", where *bindings* are "extensions of OWL interpretations that also map variables to elements of the domain".

An antecedent and a consequent of a SWRL rule consist of a conjunction of atoms. If $a_1, ..., a_n$ is a conjunction of SWRL atoms, then $\mathcal{K}_\mathcal{V} \models a_1, ..., a_n$ iff for all $i = 1...n$ $\mathcal{K}_\mathcal{V} \models a_i$.

The satisfaction of SWRL atoms is defined by means of a conditions table (see [8], Section 3), which does not contain semantics of SWRL class atoms with OWL descriptions. In order to prove the correctness of the R2ML formulas mapping into SWRL class atoms, we provide a more detailed definition of the SWRL semantics.

Let classAtom(C, t) be a SWRL class atom, where C is an OWL class description and t is either an object variable or an individual name.

1. If $C :=$ intersectionOf($c_1, ..., c_n$), where $c_1, ..., c_n$ are OWL class descriptions, then $\mathcal{K}_\mathcal{V} \models$ classAtom(C, t) iff for all $i = 1...n$ $\mathcal{K}_\mathcal{V} \models$ classAtom(c_i, t)
2. If $C :=$ unionOf($c_1, ..., c_n$), where $c_1, ..., c_n$ are OWL class descriptions, then $\mathcal{K}_\mathcal{V} \models$ classAtom(C, t) iff $\mathcal{K}_\mathcal{V} \models$ classAtom(c_1, t) or ... or $\mathcal{K}_\mathcal{V} \models$ classAtom(c_n, t)
3. If $C :=$ restr(p, someValuesFrom(CD)), where p is an individual property and CD is an OWL class description, then $\mathcal{K}_\mathcal{V} \models$ classAtom(C, t) iff for all t exists $y \in CD$ and $a \in \mathcal{O}$ such that $\mathcal{K}_{\mathcal{V}[y/a]} \models$ indPropAtom($prop, t, y$) and $\mathcal{K}_{\mathcal{V}[y/a]} \models$ classAtom(CD, y)
4. If $C :=$ restr(p, allValuesFrom(CD)), where p is an individual property and CD is an OWL description, then $\mathcal{K}_\mathcal{V} \models$ classAtom(C, t) iff for all t, for all $y \in CD$, and all $a \in \mathcal{O}$ such that $\mathcal{K}_{\mathcal{V}[y/a]} \models$ indPropAtom($prop, t, y$) implies $\mathcal{K}_{\mathcal{V}[y/a]} \models^t$ classAtom(CD, y)
5. If $C :=$ restr(p, maxCard(n)), where p is an individual property and n is a minimum cardinality, then $\mathcal{K}_\mathcal{V} \models$ classAtom(C, t) iff exists $a_1, ..., a_m \in \mathcal{O}, m < n$ such that for all $i = 1...m$ $\mathcal{K}_{\mathcal{V}[y/a_i]} \models$ classAtom(C, t), where y is a variable of the range class of p.

3.4.2 Mapping R2ML Vocabulary into OWL Vocabulary

Web Ontology Language OWL [5] is used to specify vocabularies for SWRL rules. In this section we present an abstract syntax and semantics of an OWL vocabulary. Then, following the same principles as in OCL data models mapping into R2ML, we define the interpretation mapping T from the R2ML vocabulary into OWL. The proof of faithfulness of this mapping, in the sense of preserving the satisfaction of basic atoms, is similar to the faithfulness of the OCL to R2ML mapping (see Section 3.3.2). Finally, we define the mapping of complex R2ML formulas into SWRL class atoms with OWL descriptions.

Definition 8 (OWL Vocabulary)**.** *An* OWL vocabulary *is defined by the following tuple*

$$\mathcal{V} = (\mathrm{V}_C, \mathrm{V}_D, \mathrm{V}_{IP}, \mathrm{V}_{DP}, \mathrm{V}_I, \mathrm{V}_{DL})$$

where

1. V_C *is the set of class names;*
2. V_D *is the set of datatype names;*
3. V_{IP} *is the set of individual-valued property names;*
4. V_{DP} *is the set of datatype-valued property names;*
5. V_I *is the set of individual names;*
6. V_{DL} *is the set of data literals;*

We define an abstract interpretation of OWL vocabulary, which is based on the official direct model-theoretic semantics of OWL, specified in [5], Section 3. For uniformity with previous sections of the paper we use the same symbols for denoting the set of all objects $\mathcal{O}$ and the set of all data values $\mathcal{D}$.

Definition 9 (Abstract OWL interpretation)**.** *An abstract OWL interpretation is a tuple of functions*

$$\mathcal{K} = (\mathcal{K}_{Pred}, \mathcal{K}_{Prop}, \mathcal{K}_{Ind}, \mathcal{I}_{DL})$$

such that:

1. $\mathcal{K}_{Pred}$ *maps each OWL predicate (class or datatype) into a relation;*
   (a) $\mathcal{K}_C$ *maps each OWL class c into a set of objects,* $\mathcal{K}_C(c) \in \mathcal{O}$*;*
   (b) $\mathcal{K}_D$ *maps each OWL datatype d into a set of literals,* $\mathcal{K}_D(d) \in \mathcal{D}$*;*
2. $\mathcal{K}_{Prop}$ *maps each OWL property p into a subset or a function;*
   (a) $\mathcal{K}_{IP}$ *maps each OWL individual property p into a subset, such that* $\mathcal{K}_{IP}(p) \in \mathcal{O} \times \mathcal{O}$*;*
   (b) $\mathcal{K}_{DP}$ *maps each OWL datatype property dp into a subset, such that* $\mathcal{K}_{DP}(dp) \in \mathcal{O} \times \mathcal{D}$*;*

(c) *$\mathcal{K}_{IP}$ maps each OWL individual functional property p into an object function, $\mathcal{K}_{IP}(p) \in \mathcal{O} \longrightarrow \mathcal{O}$;*

(d) *$\mathcal{K}_{DP}$ maps each OWL datatype functional property dp into a datatype function, $\mathcal{K}_{DP}(dp) \in \mathcal{O} \longrightarrow \mathcal{D}$;*

3. *$\mathcal{K}_{Ind}$ maps each OWL individual name ind into an object, $\mathcal{K}_{Ind}(ind) \in \mathcal{O}$;*

4. *$\mathcal{K}_{DL}$ maps each data literal dl into a data value, $\mathcal{K}_{DL}(dl) \in \mathcal{D}$;*

Obtaining an OWL Vocabulary from R2ML Vocabulary

In this section we explain how an OWL vocabulary can be obtained from the R2ML vocabulary, i.e. how the symbol set $\mathcal{V}oc$ of R2ML vocabulary is mapped into $\mathcal{V}$ of OWL vocabulary. We define $\mathcal{V}$ such that

1. $\mathrm{V}_C = \mathrm{OPred}^{(1)}$, i.e. the set of OWL classes is equal to the set of R2ML unary object predicates;

2. $\mathrm{V}_D = \mathrm{DPred}^{(1)}$, i.e. the set of OWL datatypes is equal to the set of R2ML unary datatype predicates;

3. $\mathrm{V}_{IP} = \mathrm{OProp}$, i.e. the set of OWL individual properties is equal to the set of R2ML object properties;

4. $\mathrm{V}_{DP} = \mathrm{DProp}$, i.e. the set of OWL datatype properties is equal to the set of R2ML datatype properties;

5. $\mathrm{V}_{Ind} = \mathrm{Obj}$, i.e. the set of OWL individual names is equal to the set of R2ML object names;

6. $\mathrm{V}_{DL} = \mathrm{DLit}$, i.e. the set of OWL data literals is equal to the set of R2ML data literals;

Since OWL and R2ML use URIs for denoting classes, properties, and datatypes, no additional mapping for names is required.

Mapping of R2ML Vocabulary Interpretations into OWL Vocabulary Interpretations

Similar to the definition of interpretations mapping from UML to R2ML, we define a mapping T from the interpretation of R2ML vocabulary into the interpretation of OWL vocabulary.

Definition 10. *The mapping T from $\mathcal{I}$ into $\mathcal{K}$ is defined as following:*

1. *$T(\mathcal{I}_{OPred^{(1)}}) = \mathcal{K}_C$ such that for all $c \in \mathrm{V}_C$, $\mathcal{K}_C(c) = \mathcal{I}_{OPred^{(1)}}(c)$, where $OPred^{(1)}$ is a set of R2ML unary object predicates;*

2. *$T(\mathcal{I}_{DPred^{(1)}}) = \mathcal{K}_D$ such that for all $d \in \mathrm{V}_D$, $\mathcal{K}_D(d) = \mathcal{I}_{DPred^{(1)}}(d)$, where $DPred^{(1)}$ is a set of R2ML unary datatype predicates;*

3. $T(\mathcal{I}_{OProp}) = \mathcal{K}_{IP}$ *such that for all* $r \in \mathrm{V}_D P$, $\mathcal{K}_{IP}(r) = \mathcal{I}_{OProp}(r)$, *where OProp is a set of R2ML object properties;*

4. $T(\mathcal{I}_{DProp}) = \mathcal{K}_{DP}$ *such that for all* $attr \in Attr$, $\mathcal{I}_{DProp}(attr) = \mathcal{J}_{Attr}(attr)$;

5. $T(\mathcal{I}_{Obj}) = \mathcal{K}_{Ind}$ *such that for all* $o_{name} \in \mathrm{V}_{Ind}$, $\mathcal{K}_{Ind}(o_{name}) = \mathcal{I}_{Obj}(o_{name})$;

6. $T(\mathcal{I}_{DLit}) = \mathcal{K}_{DL}$ *such that for all* $l \in \mathrm{V}_{DL}$, $\mathcal{K}_{DL}(l) = \mathcal{I}_{DLit}(l)$;

Mapping Basic R2ML Atoms into SWRL

We call R2ML atoms *basic atoms*, if they are constructed using either variables, or object names, or data literals as terms. The correctness of basic atoms mapping is evident and needs no proof.

Such complex R2ML terms as reference property function term, attribute function term, datatype function term, and data operation term cannot directly be mapped into SWRL since they are defined recursively (see Section 2.2.2 for definition of R2ML terms), while SWRL supports only variables and individual names as terms.

We first explain how R2ML atoms with complex terms are mapped into SWRL. In order to map R2ML atoms with **reference property function terms** and **attribute function terms**, these atoms have to be linearized, i.e. replaced by a conjunction of atoms, where each term is either a variable or a constant. The algorithm of linearization and its correctness in the sense of preserving satisfiability is presented in the literature ([47], [89]). Therefore, we do not describe it here in details, but only give an example. For instance, if $p_1...p_n.d = a$ is an R2ML attribution atom, where $p_1...p_n$ is a recursive reference property function term, then it is equal to a conjunction of two atoms: $p_1...p_{n-1} = y_{n-1} \wedge y_{n-1}.d = a$, where y_{n-1} is an object variable.

Since OWL/SWRL does not support user-defined functions, but only built-ins, we map R2ML atoms with **datatype function terms** into a conjunction of SWRL built-in atoms [47].

In further discussion we assume that object terms in all R2ML atoms are either variables or constants, i.e. we define the mapping only for basic atoms.

We consider the mapping of R2ML atoms into SWRL atoms:

- If ObjClAt(class(C), term(e)) is an R2ML object classification atom, where C is a class name and e is an object term, then it is mapped into SWRL class atom:

$$t(\text{ObjClAt(class}(C)\text{, term}(e))) := \text{classAtom}(C, e)$$

- If RefPropAt(r, subj(e), obj(f)) is an R2ML reference property atom and e, f are object terms, then it is mapped into SWRL individual property atom:

$$t(\text{RefPropAt}(r\text{, subj}(e)\text{, obj}(f))) := \text{indPropAtom}(r, e, f)$$

- If $A :=$ AttrAt($attr$, subj(e), dataVal(f)) is an R2ML attribution atom, e is an object term, and f is either a data literal or a data variable, then it is mapped into SWRL datavalue property atom:

$$t(A) := \text{datavalPropAtom}(attr, e, f)$$

- If $A :=$ DPredAt(swrlb:greaterThan, dataArgs(e, f) is an R2ML datatype predicate atom, e and f are data literals or data variables, then it is mapped into SWRL built-in atom:

$$t(A) := \text{builtinAtom(swrlb:greaterThan, } e, f)$$

  The mapping for other built-ins is similar.

- If EqAt(e, f) is an R2ML equality atom, e and f are object terms, then it is mapped into SWRL equality atom:

$$t(\text{EqAt}(e, f)) := \text{sameAs}(e, f)$$

  The mapping for inequality atom is similar.

The Faithfulness of the Interpretation Mapping T

We formulate the faithfulness theorem, which justifies the definition of the interpretations mapping T.

Theorem 3 (Faithfulness of the interpretation mapping).
Let t be the mapping of basic atoms from R2ML to SWRL and T is the mapping of interpretations as defined above. T is faithful in the sense that it preserves the satisfaction of basic atoms, i.e. the following holds:

$$\forall c \in \text{OPred}^{(1)} \text{ and variable or object name } x,\ \mathcal{I} \models \text{ObjClAt(class}(c), \text{term}(x)) \Longrightarrow$$

$$T(\mathcal{I}) \models t(\text{ObjClAt(class}(c), \text{term}(x)))$$

$$\forall r \in \mathcal{I}_{OProp} \text{ and variable or object names } x, y,\ \mathcal{I} \models \text{RefPropAt}(r, \text{subj}(x), \text{obj}(y)) \Longrightarrow$$

$$T(\mathcal{I}) \models t(\text{RefPropAt}(r, \text{subj}(x), \text{obj}(y)))$$

$$\forall attr \in \mathcal{I}_{DProp} \text{ and variable or object name } x, \text{ and data literal } d,$$
$$\mathcal{I} \models \text{AttrAt}(attr, \text{subj}(x), \text{dataVal}(d)) \Longrightarrow$$
$$T(\mathcal{J}) \models t(\text{AttrAt}(attr, \text{subj}(x), \text{dataVal}(d)))$$

The proof of the theorem is by analogy with the proof of Theorem 1.

3.4.3 Mapping R2ML Integrity Rules into SWRL Rules

A SWRL rule is an expression in the form of a logical implication. Therefore, if an OCL invariant has a form of implication, then it is mapped via R2ML integrity rule with implication into a SWRL rule with non-empty antecedent and consequent. Let

$$IR := \text{AIR}(\text{constr}(\text{UnivQuantF}(\text{vars}(x), \text{Impl}(\text{cons}(e_1(x)), \text{antec}(e_2(x))))))$$

be an R2ML integrity rule with an implication, then

$$t(IR) := \text{Implies}(\text{Antec}(t(e_2(x))), \text{Conseq}(t(e_1(x))))$$

If an original OCL invariant and, consequently, corresponding R2ML integrity rule, is not an implication, then it corresponds to the SWRL rule with an empty antecedent.

A conjunction of R2ML atoms is generally mapped into a conjunction of SWRL atoms, except one case: a conjunction of R2ML classification atoms with the same variable is mapped into one SWRL class atom with OWL IntersectionOf description. The mapping for this case is defined below.

Mapping R2ML Formulae into SWRL Class Atoms with OWL Descriptions

Complex R2ML logical formulas like existentially and universally quantified formulas are mapped into SWRL class atoms with OWL descriptions. Not all R2ML logical formulas can be mapped into OWL/SWRL since some formulas cannot be represented in SWRL by means of an OWL description.

- If $F :=$ ExQuantF(vars(y) And(RefPropAt(*prop*, subj(x), obj(y)), C))
  is an R2ML existentially quantified formula, where $LF(y)$ is a logical formula, then

  $$t(F) := \text{classAtom}(\text{restr}(prop, \text{someValuesFrom}(t(LF(y))), x)$$

- If $F :=$ UnivQuantF(vars(y) And(RefPropAt(*prop*, subj(x), obj(y)), LF))
  is an R2ML universally quantified formula, where $LF(y)$ is a logical formula, then

  $$t(F) := \text{classAtom}(\text{restr}(prop, \text{allValuesFrom}(t(LF(y))), x)$$

- If

  $$\begin{aligned} F \quad := \quad & \text{AtLeastQuanF}(\text{vars}(\text{OVar}(y)), \text{minCard}(n), \\ & \text{RefPropAt}(prop, \text{subj}(x), \text{obj}(y))) \end{aligned}$$

  is an R2ML at least quantified formula, then

  $$t(F) := \text{classAtom}(\text{restr}(prop, \text{maxCardinality}(n)), x)$$

- If $F := \text{And}(LF_1(x), LF_2(x))$ is a conjunction of R2ML formulas, where $LF_1(x)$ and $LF_2(x)$ are logical formulas with a universally quantified variable x, then

  $$t(F) := \text{classAtom}(\text{IntersectionOf}(t(LF_1), t(LF_2)), x)$$

- If $F := \mathrm{Or}(LF_1(x), LF_2(x))$ is a disjunction of R2ML formulas, where $LF_1(x)$ and $LF_2(x)$ are logical formulas with a universally quantified variable x, then

$$t(F) := \mathrm{classAtom}(\mathrm{UnionOf}(t(LF_1), t(LF_2)), x)$$

In the SWRL syntax for class atoms with OWL descriptions some variables are not explicitly specified, while in OCL and R2ML they are explicit. For instance, an expression "at least one maker of a wine is a winery" in OCL is

$$\mathrm{Inv}(\mathrm{Wine}, x.\mathrm{hasMaker} \rightarrow \mathrm{exists}(y|y.\mathrm{oclIsKindOf}(\mathrm{Winery})))$$

while in SWRL it is

$$\mathrm{Implies}(\mathrm{Antec}(\mathrm{classAtom}(\mathrm{Wine},\ \mathrm{x})),$$

$$\mathrm{Conseq}(\mathrm{classAtom}(\mathrm{restr}(\mathrm{hasMaker},\ \mathrm{someValFrom}(\mathrm{Winery})),\ x)))$$

The variable y, which is explicit in the OCL expression, is implicit in the SWRL expression. Therefore, in order to obtain a syntactically correct SWRL expression we have to count the context of the formula and drop some explicit variable names while mapping from OCL and R2ML. For instance, if F is an R2ML formula at the outermost level of a rule, then it is mapped as described above; if F is a sub-formula of another formula (like LF_1 and LF_2 in the above mapping), then it is mapped directly into an OWL class description as follows:

- If $F :=$ ExQuantF(vars(y) And(RefPropAt($prop$, subj(x), obj(y)), LF))
  is an R2ML existentially quantified formula, where LF is a logical formula, then

$$t(F) := \mathrm{restr}(prop, \mathrm{someValuesFrom}(t(LF))$$

- If $F :=$ UnivQuantF(vars(y) And(RefPropAt($prop$, subj(x), obj(y)), LF))
  is an R2ML universally quantified formula, where LF is a logical formula, then

$$t(F) := \mathrm{restr}(prop, \mathrm{allValuesFrom}(t(LF))$$

- If $F := \mathrm{And}(LF_1(x), LF_2(x))$ is a conjunction of R2ML formulas, where $LF_1(x)$ and $LF_2(x)$ are logical formulas with a universally quantified variable x, then

$$t(F) := \mathrm{IntersectionOf}(t(LF_1), t(LF_2))$$

- If $F := \mathrm{ObjClAt}(C, x)$ is an R2ML object classification atom, where C is a class name and x is an object variable, then

$$t(F) := C$$

An R2ML to SWRL Mapping Example

Let us consider an expression "*At least one maker of a wine is a winery*". This expression corresponds to the following OCL invariant Inv(Wine, R) where

$$R := x.\text{hasMaker} \rightarrow \text{exists}(y|y.\text{oclIsKinOf}(\text{Winery}))$$

We map the OCL invariant into an R2ML integrity rule:

$$\begin{aligned} t(\text{Inv}(\text{Wine, R})) \quad := \quad & \text{AIR}(\text{constr}(\text{UnivQuantF}(\text{vars}(x), \\ & \text{Impl}(\text{antec}(\text{ObjClAt}(\text{class}(\text{Wine}))), \text{cons}(\text{t}(\text{R})))))) \end{aligned}$$

The result of the R2ML integrity rule mapping into a SWRL implication:

$$\text{Implies}(\text{Antec}(\text{classAtom}(\text{Wine, x})), \text{Conseq}(\text{t}(\text{R})))$$

And $t(R)$ from OCL to R2ML is as following:

$$\begin{aligned} t(R) \quad := \quad & \text{ExQuantF}(\text{vars}(y), \\ & \text{And}(\text{RefPropAt}(\text{hasMaker, subj}(x), \text{obj}(y)), \\ & \quad t(y.\text{oclIsKindOf}(\text{Winery})))) := \\ & \text{ExQuantF}(\text{vars}(y), \\ & \text{And}(\text{RefPropAt}(\text{hasMaker, subj}(x), \text{obj}(y)), \\ & \quad \text{ObjClAt}(\text{Winery, } y)))) \end{aligned}$$

And then the mapping from R2ML into SWRL:

$$\begin{aligned} & t(\text{ExQuantF}(\text{vars}(y), \\ & \quad \text{And}(\text{RefPropAt}(\text{hasMaker, subj}(x), \text{obj}(y)), \\ & \quad\quad \text{ObjClAt}(\text{Winery, } y)))) := \\ & \quad \text{classAtom}(\text{restr}(\text{hasMaker, someValueFrom}(t(\text{ObjClAt}(\text{Winery, } y))))) := \\ & \quad \text{classAtom}(\text{restr}(\text{hasMaker, someValueFrom}(\text{Winery})), x) \end{aligned}$$

The Correctness of the R2ML Integrity Rules Mapping into SWRL

Theorem 4 (Semantical correctness of R2ML to SWRL mapping).
The mapping t of R2ML integrity rules into SWRL rules is *semantically correct*, i.e. it preserves the satisfaction of rules (see Definition 5).

Proof. The proof is similar to the proof of Theorem 2: we consider mapping of basic atoms as a basis and then prove correctness of more complicated expressions. Let us prove here the correctness of the mapping for R2ML conjunction. We assume that the implication holds for formulae $f_1, ..., f_n$, so that for all $i = 1...n$

$$\mathcal{I}_\mathcal{V} \models f_i \Longrightarrow T(\mathcal{I}_\mathcal{V}) \models t(f_i)$$

.

We have to prove that

$$\mathcal{I}_{\mathcal{V}} \models \text{And}(f_1, ..., f_n) \Longrightarrow T(\mathcal{K}_{\mathcal{V}}) \models t(\text{And}(f_1, ..., f_n)) \tag{3.8}$$

This is equivalent to

$$\mathcal{I}_{\mathcal{V}} \models \text{And}(f_1, ..., f_n) \Longrightarrow \mathcal{K}_{\mathcal{V}} \models \text{classAtom}(\text{IntersectionOf}(t(f_1), ..., t(f_n)), x)$$

where $\mathcal{K}_{\mathcal{V}} = T(\mathcal{I}_{\mathcal{V}})$.

According to the definition of the satisfaction relation for SWRL IntersectionOf,

$$\mathcal{K}_{\mathcal{V}} \models \text{classAtom}(\text{IntersectionOf}(t(f_1), ..., t(f_n)), x)$$

iff for all $i = 1...n$

$$\mathcal{K}_{\mathcal{V}} \models \text{classAtom}(t(f_i))$$

On the other hand, the left-hand side of the implication (3.8) holds iff for all $i = 1...n$

$$\mathcal{I}_{\mathcal{V}} \models f_i$$

Consequently, implication (3.8) holds by induction hypothesis: for all $i = 1...n$,

$$\mathcal{I}_{\mathcal{V}} \models f_i \Longrightarrow T(\mathcal{I}_{\mathcal{V}}) \models t(f_i)$$

□

3.5 Limitations and Conclusions

We highlight two major contributions of the work:

- The mapping from OCL invariants into SWRL rules via the rule interchange format R2ML;
- The proof of the correctness of this mapping in the sense that it preserves the satisfaction of formulae.

The result of the work is useful for various communities. Software developers, who actively use UML/OCL, may employ the interchange mapping in order to translate their rules to SWRL and use them in a Semantic Web application. UML tool vendors can extend their tools with a rule interchange functionality. As a prototype, we refer to UML-based rule modeling tool Strelka with the rule interchange support[3]. On the other hand, the work on rule interchange is interesting for Semantic Web practitioners, who work in the area of formal semantics of rule languages and have interest in the rule interchange standardization.

The research can be reused in following ways:

[3]Strelka homepage: `http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=Strelka`

- Since OCL to SWRL interchange is performed via R2ML and consists of two parts, each part can be substituted with an additional mapping, while leaving other part untouched. This allows quick integration of mappings with modeling tools and Web services. For instance, Strelka implements a mapping from a special graphical notation for rules into R2ML, while a web-service supports different mappings from R2ML into such rule languages as Jess, F-Logic and JBoss Rules. More on this is available in [50].

- The formal problem, specified in Section 3.2, is solved for two particular rule languages: OCL and SWRL. However, the presented solution can be adopted for other rule languages, which syntax and semantics are defined.

- When W3C will release the final RIF standard, the presented approach to the semantic correctness can be used by rule vendors in order to verify translators from their rule language into RIF.

The main limitation of the conducted research concerns the subset of OCL for which the mapping is defined. The expressivity of OCL is higher than the first order logic since closures over object relations can be expressed in OCL but not in FOL [54]. Therefore, OCL is more expressive than SWRL and we need a subset of OCL, which can be mapped into SWRL. Such a subset, used in this paper, is defined in [48] and, in particular, does not support OCL collection operations. Datalog languages like SWRL without collections support are still capable of representing a variety of business rules. For instance, this is proved by the case study [34], where the Object-Role Modeling (ORM) methodology [32], which does not support collections, is used to model business rules in the domain of e-commerce complaints. Another rule case study in the area of financial services is modeled in UML using production rules and OCL expressions without collections. These rules then translated into Jena Rules and JBoss Rules [94].

We conclude with a list of possible issues for the further research:

- The mapping from SWRL to OCL and its correctness can be useful if some part of an OWL ontology and SWRL rules need to be remodeled in UML/OCL;

- The mapping from OCL derivation rules into some other rule language, for instance SQL, where derivation rules are represented by means of materialized views can be useful for the software engineering and database communities;

- The issue of OCL collections mapping needs further research on semantics.

Chapter 4

Production Rule Verification

4.1 Introduction

Since production rules are widely used in industry and supported by different vendors such as Oracle, ILOG, JBoss and others, the problem of rule quality is important.

It is widely recognized that the process of verification is an important part of quality assurance for rule systems. Even if the definition of the term "verification" varies in the literature, we use the common definition, also used in [73], which is that verification of rules includes checking the knowledge base for logical anomalies such as redundant rules, contradictory rules, and missing rules. Such verification is called anomaly detection. It is important to note that mentioned anomalies are not necessarily errors, but rather potential errors, also known as "attention points ... that might give hints for incorrect rules or wrongly expressed knowledge"[85].

Some of the existing systems provide built-in verification capabilities. For instance, JBoss has recently implemented a rule-based verifier, which is a part of JBoss Drools version 5.0. On the other hand, Semantic Web rule systems such as Jena do not have verification components yet. We believe that due to the growth of interest in Semantic Web technologies by companies and organizations, the need for tools and methodologies, which check and improve the quality of rules for the Semantic Web, will rise. An analogy here is the progress of software development methodologies, where the role of testing, aimed to improve quality, cannot be underestimated. Rule systems for the Semantic Web with modeling tools and means of rule quality control are relatively new and a lot of research on quality of rule-based applications is currently in progress and will be conducted in the future. In this work, we contribute to the research on rule verification by providing a verification approach for Jena rules, which are actively used in academic as well as in industrial research and development.

As it is defined in Section 1.1.1, verification does not check whether rules under consideration achieve a correct business goal expected by business people, but checks whether rules are logically consistent and complete. "The process that aims for the detection of incorrect results or undesired (business) behavior" [30] of rules is called *validation*. In this work, we provide a solution for the verification of production rules, but since validation and verification are often considered jointly, we give a brief overview of how validation works. As it is stated in [30], "the most common way of rules validation is to just pass the

(changed) rules to another member of organization." It means that validation is mainly a manual process, where a business expert analyzes the rules and decides whether their execution results are as expected. However, the human involvement in validation can be reduced [44] and there are various validation approaches and frameworks for validation of rule-based systems ([40], [43], [39]).

Verification can be implemented either by means of algorithms (see Section 4.3 for the overview of related works) or declaratively by means of rules.

Most of the existing verification approaches are algorithm-based. In this work, we provide a declarative approach to the verification of knowledge bases, which contain production rules and integrity constraints.

We distinguish between business rules, which have to be verified, and verifier rules, which are used for verifying of business rules.

A *verifier rule* is a production rule, which is executed when a business rule base contains an anomaly and generates a report about it.

A *rule-based verification approach* employs verifier rules for detecting anomalies in business rules.

The declarative verification approach has a number of advantages:

- Simplicity of implementation. Anomalies are described by means of special rules, called verifier rules, which are executed by a rule engine. It means that rule-based verification is about writing verifier rules, which is easier than developing and implementing algorithms.

- Easiness of maintenance. When new anomalies are discovered, new verifier rules can be added easily in order to detect them. No additional anomaly-detection programming and algorithm development is required.

- Support for various rule languages. Verifier rules are expressed in terms of a generic rule metamodel, which, if general and flexible enough, can be used for various rule languages. In this thesis we provide the generic metamodel for Jena rules and discuss some related works, but do not consider the issue of the most general metamodel for all rule languages.

Our verification approach follows general principles, which are, in particular, formulated in [73] and [14]:

- Anomalies are detected by examining the syntax of a knowledge base (KB), although they may be understood semantically. This means, for example, where there are two identical rules in the knowledge base, one of them is clearly redundant semantically, but this can be detected only by finding two rules that are syntactically equal.

- Anomaly detection methods apply only to the knowledge base and certain properties of the rule engine are assumed but not verified.

- Anomalies are not necessarily errors, but they are symptoms of probable errors in the knowledge base.

The main objective of this research is to employ the rule-based verification approach for verification of Jena rules. Jena is a Semantic Web framework, which is widely used in academia and industry for developing rule-based applications, mainly for the Semantic Web. We introduce Jena in Section 4.6.1.

The main idea of the rule-based verification approach is that anomalies, defined in Section 4.5.1, are described by means of verifier rules. These rules are metarules in the sense that they do not solve particular tasks in the business domain, but they are rather "rules about rules": They use the Jena rule metamodel as a vocabulary.

We use the syntax of JBoss Rules (introduced in Section 4.6.2) for expressing verifier rules and the Drools engine to execute them. It is possible to express verifier rules for Jena using the Jena rule syntax. However, Jena syntax is verbose and for practical reasons, it is easier to use Drools. Another argument in favor of Drools as a rule engine for verifier rules is the JBoss verifier (Section 4.3.7), which is designed in a way similar to ours, but for the verification of JBoss rules. Therefore, it is natural to reuse this knowledge when solving the practical task of Jena rule verification.

In order to achieve the declared research objective, we first formally define a knowledge base with production rules and integrity constraints and introduce the execution semantics of production rules (Section 4.4). Then we define different classes of anomalies using model theory (Section 4.5). Finally, we provide verifier rules for defined anomalies (Section 4.6). All definitions are explained by means of a running example, initially presented in Section 4.2.

An overview of related works on rule verification is given in Section 4.3.

4.2 Running Example: UServ Case Study

We introduce a set of rules expressed in the natural (English) language. These rules are an excerpt from the UServ Case Study, proposed by the Business Rules Community for demonstration and testing purposes. The rule set consists of a number of production rules and semantic constraints and contains different anomalies. We refer to these rules when defining anomalies in next sections.

Rules:

Rule 1 If the driver is male and is under the age of 26, then he is a young driver.

Rule 2 If the driver is under the age of 26, then he is a young driver.

Rule 3 If the car is less than 5 years old and more than 10 years old, then increase premium by 300.

Rule 4 If a car is the luxury car, then base premium is 500.

Rule 5 If the car is less than 5 years old and is luxury car, then base premium is 300.

Rule 6 If the car's price is greater than 45 000, then the car's potential theft rating is "high".

Rule 7 If the car is a convertible, then the car's potential theft rating is "moderate".

Rule 8 If the driver is eligible and has no training certification, then he is a high risk driver.

Rule 9 If the driver is over the age of 26, then he is a normal driver.

Rule 10 If the driver is over the age of 70, then he is a senior driver.

Rule 11 If the driver is under the age of 26, then he is a young driver.

Rule 12 If the car is provisional and is older than 3 years, then it is not eligible car.

Rule 13 If the potential theft rating of a car is "high", then the car's eligibility is "provisional".

Constraints:

Constr 1 No driver is a normal driver and a senior driver.

Constr 2 No car eligibility is "provisional" and "not eligible".

Constr 3 Every eligible driver has training certification.

The rules above are expressed informally in the natural (English) language. In order to execute them in a rule application, they have to be expressed in some formal rule language. Simply by looking at the informal textual expressions it is not always possible to say what kind of a formal rule it is: a production rule, a derivation rule or an integrity constraint (see the classification of rules in [6], page 6). In order to simplify formalization tasks, there are various guidelines, standards and controlled languages for expressing business rules. For instance, SBVR [69] explains how to express business rules, depending on a purpose. The Object-Role Modeling methodology (ORM) [33] also implies some guidelines for expressing business rules by providing a verbalization mechanism for ORM models. There is a whole family of so called controlled languages, which restricts a human language and makes it formal. This simplifies the further formalization process. The most prominent example of such language is the Attempto Controlled English (ACE) [29]. Other examples are Common Logic Controlled English [84] and Metalog [55].

Since our goal is to verify Jena rules, which are production rules, we consider the examples (Rule 1 - Rule 13) above as production rules. The decision about the rule type depends on the rule application and business tasks, but does not depend on a rule language or a formalization method. For instance, if an intended business goal is to update the knowledge by means of rules, then business rules can be formalized as production rules; if the goal is to query the knowledge base for facts, then business rules can be formalized as derivation rules; and if the goal is to constraint a presence of some facts in the knowledge base, then business rules can be formalized as integrity constraints.

4.3 Related Works on Rules Verification

Some of the early approaches to verification employed condition/action tables, which separate rule's condition and action parameters. Algorithms then check for the existence of

relationships among the rows and columns. Examples of the table-based approaches are Expert System Checker (ESC) and the Rule Checking Program (RCP). The CLIPS toolset [20], a predecessor of Jena Rules, supports verification via cross referencing of parameters and relations in the rule base.

4.3.1 COVER System

Preece's COVER system ([73], [74]) improves the table-based methods by constructing a graph representation of the rule base directly from the rules. The advantage here is that anomaly detection works across numerous rules, rather than between rule pairs, as it is done with the table-based approaches.

4.3.2 Petri Nets-based Approaches

A development which attempts to address the order in which rules are executed (and thus makes less assumptions conflict resolution) is based on Petri nets ([10] and [46]). A rule base can be represented as a Petri net and is then tested for completeness using existing methods. It is also suggested that this approach can handle temporal relationships between rules. However, transformation of a rule base into a Petri net is a non-trivial task.

4.3.3 Truth Maintenance Systems

Another early work on the verification of expert system knowledge bases using truth maintenance systems is introduced in [96]. The main advantage of the verification techniques based on truth maintenance is that they either generate the sets of facts which might lead to certain anomalies, or logically deduce all explicit and implicit anomalies which the KB contains. Since the knowledge base is reformulated in terms of truth maintenance theory, the dependencies among final conclusions and sets of facts upon which these conclusions depend become explicitly clear. One step of the detection algorithm is the computation of the so-called ground stable extension of the original knowledge base, which is later examined for anomalies. The approach seems to be powerful, however it is not clear how the rules execution semantics is preserved and how complex it is in practice to calculate the ground stable extension.

4.3.4 Verification of Non-monotonic Knowledge Bases

General approaches to the verification of non-monotonic knowledge bases is presented in ([97], [14]). These works extend algorithms, defined for monotonic knowledge bases, using the concept of a default rule. Proposed algorithms mostly give a theoretical foundation for algorithms which verify non-monotonic knowledge bases with production rules. The R2ML being a general rule language, supports default rules by means of negation as failure. However, the number of production rule systems which support non-monotonic reasoning is small and widely used production rule systems interpret negation as a simple absence of a fact in the working memory.

4.3.5 Term Rewrite Semantics for Rule Verification

A recent work on detecting redundancies in production rules is based on term rewrite semantics [82]. The research presents a sound, but incomplete algorithm, which may detect wide classes of redundant rules. The advantage of this method is that it generalizes some previously defined pattern-based approaches to redundancy detection by limiting the depth of the search. However, the presented approach is defined for redundancy only, which is important, but not the only type of possible anomalies in production rule bases.

4.3.6 Test-based Approaches to Rules Verification

There is a work on rules validation by means of test cases [72], which describes how to check whether a rule base produces expected results. The detection of some basic anomalies by means of test-cases is presented in [25]. The approach adopts test-driven software development techniques from the software engineering community. Using the same test-driven approach in [72] it is discussed how to check whether a rule base produces expected results by defining tests for rules. The work is mostly related to the rule validation, since it it is aimed at detection of incorrect results or undesired (business) behavior.

4.3.7 Other Approaches

An on-line verification of tabular rule-based systems with the eXtended Tabular Tree (XTT) is presented in [45]. A visual edition tool "Mirella" [63] for design and verification employs the XTT method and implements the engine for analysis and verification of such rule base properties as completeness, determinism and subsumption.

The Drools 5.0 contains the Rule Analytics Module [79], which employs rule-based verification for verifying JBoss rules. The work on the module is in progress and the amount of anomalies which can be detected is growing.

4.4 Knowledge Bases with Production Rules

In this section, we give the definition of a knowledge base with production rules and integrity constraints. In addition, the operational semantics of production rules is presented in order to give readers the basic understanding of production rule engines. We also discuss how our production rule representation corresponds to latest works of W3C and OMG concerning production rules standardization.

4.4.1 Production Rules

A production rule is used for specifying an action, which is performed if a rule condition is satisfied. An example of a production rule:

> If the driver is a young driver, then give him a premium of $300.

In this rule, "*the driver is a young driver*" is a rule condition and "*give him a premium of $300*" is an action.

Production Rule

Definition 11 (Production rule)**.** *A* production rule *is a tuple $C \Rightarrow RHS$, where C is a literal conjunction (a conjunction of atoms or negated atoms) as a condition, and RHS is a literal conjunction as a postcondition. A condition is also called* antecedent *and the right-hand side with a postcondition is also called* consequent*.*

If r is a production rule, then $cond(r)$ denotes the condition of r and $pcond(r)$ denotes the postcondition of r.

In this definition the condition and the postcondition are literal conjunctions, while in some rule frameworks, for instance in R2ML, these are the most general quantifier free logical formulas with weak and strong negations. Since our final goal is to verify Jena rules, we use this simplified definition which is very close to the structure of rules in Jena.

As it is stated in the RIF-PRD [77], the condition of a production rule "is like the condition part of logic rules, as covered by the basic logic dialect of the W3C rule interchange format, RIF-BLD [75]." Therefore, the intended semantics of the condition part of a production rule can be given using the model theory. RIF-PRD, for instance, explains the compatibility of the condition part of production rules with RIF-BLD. However, "production rules, in general, are not logic rules, and they are not amenable to a model theory" [77]. Their intended semantics is specified operationally in Section 4.4.4. Using RIF-PRD terminology for actions, logical atoms, which represent postconditions, can be viewed as as *targets* of actions in the sense that the postcondition of a rule must hold when the rule is executed. In other words, postconditions specify the execution effect of the rule.

A Production Rules Representation by OMG

OMG released a beta of the standard for production rules representation [6], which supposes to fulfill a number of requirements related to business rules, software systems and other rule standards. "It provides a standard production rule definition that supports and encourages system vendors to support production rule execution" ([6], Scope) and can also be used for rule interchange amongst rule modeling tools. The OMG PRR defines a production rule as "a statement of programming logic that specifies the execution of one or more actions in the case that its conditions are satisfied" of the form:

> if [condition] then [action-list]

The PRR is defined by means of two metamodels, defined by means of MOF/UML: PRR Core metamodel and PRR OCL - non-normative abstract OCL-based syntax for PRR expressions, defined as an extension of PRR Core metamodel. The PRR Core "is a set of classes that allow for production rules and rulesets to be defined in a purely platform independent way without having to specify OCL to represent conditions and actions."

The rule action part defines an ordered list of actions. The OMG PRR [6] defines the following state changing actions:

Update action expression is used to specify a new value for a property. It consists of an object term as context argument and refers to a property, which is updated by the action.

Assert action expression is used to create an object. It consists of an object term as a context argument, a list of slots as parameters to the object constructor and it refers to a class, whose object creation is specified by the action.

Retract action expression is used to remove an object. It consists of an object term as a context argument and refers to a class, which object is removing by the action.

Jena 2 supports only two types of actions: Assert and retract. An update action is implemented by a sequence of retract action (remove old facts) and assert action (assert new facts). More about Jena is in Section 4.6.1.

RIF Production Rule Dialect

Within the scope of the W3C work on the Rule Interchange Format [75], the Production Rule Dialect has been proposed and a draft document has been released [77]. We mention the proposal in this section since it is a significant activity in the area of production rule standardization, though it is not directly connected to the topic of rule verification. The purpose of the dialect is to perform the interchange of production rules. The document defines the common XML syntax for production rules, which is compatible with RIF-BLD [75] in the condition part of a production rule. A special syntax for production rule actions is also defined. The semantics of production rules is given in two parts: Semantics of condition formulae, using model theory and the operational semantics, using the transition relation.

The correspondence between specification of action types by the OMG PRR proposal [6] and actions, defined by RIF-PRD, is not considered in the current draft of RIF-PRD.

The document considers running rule examples, which are originally expressed in English and then are encoded using RIF-PRD XML syntax. However, there are no examples of production rules, expressed in other rule languages and it is not shown how RIF-PRD is capable of encoding these rules and interchanging them. Actually, the issue of interchange transformations is not yet covered by the current draft.

The W3C RIF overlaps with the OMG PRR in scope and they "share the common goal of of rule interoperability, albeit for different stages of the software development lifecycle" ([6], Annex D). The differences are as following:

- OMG PRR focuses on modeling of production rules with an XMI-compliant interchange format for UML-based modeling tools.
- W3C RIF focuses on an interchange format for Web rules and for web-developers, which mainly use XML-based web technologies.

There are joint work plans between the W3C working group and PRR group, which have a considerable overlap in membership. In particular, the RIF working group is going to employ the PRR Core metamodel to maximize the value of these standards efforts in both groups. The RIF working group is encouraged to produce a document showing how W3C RIF standard works together with OMG PRR. There are plans for an extension of a future version of PRR with a W3C RIF syntax for conditions and actions of production rules.

4.4.2 Rule Vocabulary and Semantic Constraints

According to the Business Rules Manifesto [80], rules are built on vocabularies as expressed by terms. Since a business domain normally has a natural vocabulary, which defines business concepts and some relations among them, a rule system should also support vocabularies. Modern rule systems, like Oracle Business Rules, ILOG, JBoss and Jena 2 express vocabularies using various vocabulary representation languages. Building of a correct rule application starts with a definition of a vocabulary. For instance, OCL uses UML class models as a vocabulary. UML classes form a type system for OCL expressions. The Semantic Web Rule Language (SWRL) uses Web Ontology Language (OWL) as a vocabulary language. Jena rules may use RDF Schema or OWL for expressing vocabularies.

Usually, vocabulary languages have formal semantics and in addition to a simple list of concepts and relations, may express constraints, also called integrity rules. The term "business vocabulary" comes mainly from the business rules community. In the further discussion, we separate a rule vocabulary, which simply defines concepts, from integrity rules. In some works on rule verification ([73], [74]), such rules are also called *semantic constraints* and defined as set of literals with the intended meaning that its conjunction is logically inconsistent.

However, in our approach semantic constraints have more complicated structure, since they are obtained from OWL axioms. We define the structure of these constraints in Section 4.6.10.

4.4.3 Knowledge Base

Definition 12 (Knowledge Base)**.** *A production rule base R together with a vocabulary V of R, semantic constraints C, and a fact set X forms a* knowledge base $KB = \langle X, V, C, R \rangle$.

The fact set X consists of a set of ground facts (variable-free atoms). It has a property that for two facts we may say which one is more recent than the other, i.e. each fact is assigned timestamp when it was asserted or updated in X. This issue is important for defining the operational semantics of production rules.

4.4.4 Operational Semantics of Production Rules

The operational semantics of production rules is described in various papers and there are different implementations: OPS5 [27], Clips [23], Jess [28], Jena [3], JRules [9], Drools [1]. Such systems are usually based on different versions of the rete algorithm, which was initially introduced in [26]. In [22], production systems and rete algorithm formalization are described. The rewrite semantics for production rule systems is presented in [83].

OMG PRR gives an informal, but extended description of production rule execution mechanism [6]. The RIF-PRD by W3C [77] also defines the operational semantics by means of the transition relation.

Further on in this section, we give the operational semantics of production rules, which is very close to what is described in RIF-PRD and to the semantics supported by the Jena forward-chaining engine.

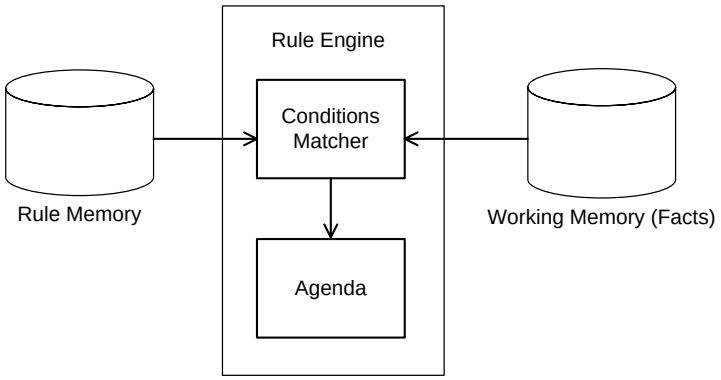

Figure 4.1: Production rules system

The operational semantics of production rules, specified below, is not directly used in our rule verification approach, but may help readers to understand how production rule systems work and what possible consequences of rule execution are.

The typical production rule system contains the following components (see Figure 4.1):

- The *Working Memory*, which consists of a set of ground facts (variable-free atoms);
- A separate *rule memory* with inference rules;
- A *condition matcher*, which computes the subset of rules whose left-hand side is satisfied by the current content of the working memory;
- A rules *agenda*, which manages the execution order of rules using a *Conflict Resolution* strategy.

The rules execution algorithm consists of three basic phases (see Figure 4.2):

- The *match phase*, where the condition matcher is employed;
- The *conflict resolution phase*, where a rule is selected for an execution after conflict resolution strategies have been applied. In our execution, model the rule is selected according to the rule selection principle (see Definition 16 below);
- The *act phase*, which is the final step in each cycle when an action of the chosen rule is executed.

Below, we define the operational semantics of production rules formally and denote how it is related to the phases of the execution model depicted in Figure 4.2.

In this work, we define the formal semantics of production rules on the basis of a high-level transition system formalism. When production rules are executed in a system, they typically change its state, which may be given by a fact set $X \subseteq L$ in the context of an information model theory $\langle L, \leq, \mathcal{I}, \models \rangle$, where L is a set of logical formulae, called a language, $\mathcal{I}$ is a set of interpretations, $\leq$ is an information ordering, which allows comparison of the information content of two fact sets $X_1, X_2 \subseteq L$: whenever $X_1 \leq X_2$, we

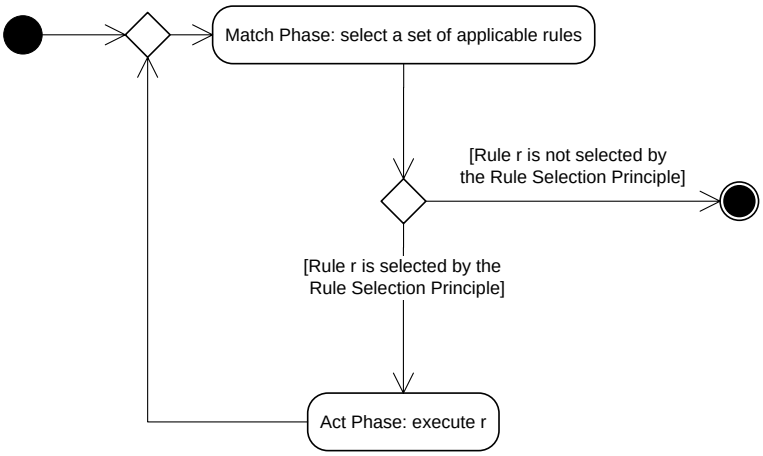

Figure 4.2: Production rules execution model

say that X_2 is more informative that X_1, and $\models$ is the satisfaction relation. We assume that such a system possesses a state change function

$$Upd : 2^L \times L \longrightarrow 2^L$$

such that $Upd(X, F)$ provides the updated knowledge state obtained from X by assimilating the input formula F, and it holds that

$$Upd(X, F) \models_* F$$

where $\models_*$ denotes the entailment relationship, based on the underlying non-standard model operator Mod_*. The non-standard model operator does not provide all models of a knowledge base, but only a special subset that is supposed to capture its intended models according to some semantics.

In the *match phase*, a subset of rules, whose left hand side is satisfied by X, is computed. This process is called unification and rules from the resulting set are called *applicable rules*.

The unification algorithm takes two sentences and returns a substitution that makes them look the same if such a substitution exists.

Definition 13 (Unification)**.** *$Unify(p, q) = \theta$, where p and q are atomic formulas and $Subst(\theta, p) = Subst(\theta, q)$. If no θ exists, then $Unify$ returns* fail*. θ is called the* most general unifier (mgu) *of the two atoms, which makes the least commitment about the bindings of the variables.*

The problem of matching a pair of literals is addressed by unification, which is an inefficient way of implementing a forward-chaining production rule system, when it is used in a straightforward way. This performance problem was addressed by the **rete** algorithm, which compiles the rule memory into the network, where duplication of rules are eliminated. A good example of the rete algorithm at work is given in [81].

Definition 14 (Applicable rule)**.** *A rule r is* applicable *to X if exists θ such that $X \vdash Subst(\theta, cond(r))$.*

Let us define an $Exec$ function, which updates X by executing an applicable rule r.

Definition 15 (Exec function)**.** *Let r be an applicable rule and θ the most general unifier for X and r, then*

$$Exec(X, \{r\}) = Upd(X, a\theta)$$

Example 1. *Let $X = \{p(a)\}$, r be a rule and $\theta = \{x = a\}$, then*

$$X_1 = Exec(X, r) = Upd(X, q(a)) = \{p(a), q(a)\}$$

Applicable rules are computed in the match phase of the algorithm and some production rule systems execute the actions of all rules that pass this phase. But in case more than one rule passes the match phase, the execution of all actions may lead to contradictions or poorer performance. Possible conflicts are resolved on the *conflict resolution phase*, which may implement some of the following strategies:

- Do not execute the same rule on the same arguments twice;
- Prefer rules that refer to recently created working memory elements;
- Prefer rules that are more specific, i.e. do not execute subsumed rules (see Section 4.5.1);
- Prefer actions with higher priority, as specified by some ranking.

Below we define an execution algorithm, which is similar to the algorithm, used in the popular production rule engines like Drools or Jena 2. When there are multiple applicable rules, the rule engine needs to know in which order rules should be executed. Rule engines have different mechanisms for defining the order. For instance, Drools has two default strategies: Salience and LIFO (last in, first out). Salience is a priority, which can be specified by users and defines which rule has a higher priority then other rules by giving a higher number. In general, we assume that rules in R are ordered according to some linear order. As we said before, the executed actions of production rules change the state of the system. We define a system state S on the i-th transition step as a tuple:

$$S_i = (X_i, R_i)$$

where $X_i \subseteq X$ is a fact set on the i-th transition step, $R_i = \{r_i^1, \ldots, r_i^n\}$ is a set of rules from R applicable to X_i. Let $F_i^j \subseteq X_i$ be the most recent minimal set of facts, such that $F_i^j \vdash cond(r_i^j)\theta$, where θ is the most general unifier. The conflict resolution strategy of unifying rule conditions with the most recent working memory facts is preserved by the definition of F_i^j. Another conflict resolution strategy of non-execution of the same rule on the same arguments twice is preserved by the following rule selection principle.

Definition 16 (Rule selection principle)**.** *Let S_i be a system state and $r_i^j \in R_i$ is a rule applicable to X_i. Then this rule is selected for the execution iff the rule was not previously applied on the same set of facts, i.e. there is no l such that $r_i^j \in R_l$ and $F_i^j = F_l^j$, $l \leq i$.*

Some strategies listed above are not counted by our model because some of them are optimization strategies and some are specific for some platforms. For instance, neither JBoss nor Jena 2 implement the non-execution of a subsumed rule if such rule is in the agenda. Execution with respect to priorities is a practical issue, which, in some cases, may simplify the programming of a rule application, however, the general guideline is not to use priorities.

Let us define a function

$$UpdState(X_n, R_n) = (Exec(X_{n-1}, r), R_n)$$

where $r \in R_{n-1}$ is the first applicable rule which is selected according to the rule selection principle.

The production rules execution algorithm is the following:

1. $S_0 = (X_0 = X, R_0)$, where $R_0 \subseteq R$ is a set of rules, applicable to X_0;
2. $S_n = UpdState(S_{n-1}, R_{n-1})$.

The execution stops when it is not possible to select a rule from R_n according to the rule selection principle, i.e. all applicable rules are executed once on the corresponding facts. An invocation of the *UpdState* function corresponds to the act phase of the algorithm. Since the rule is executed, the algorithm goes again to the match phase and calculates a new set of applicable rules.

4.5 Anomaly Definitions

According to the anomaly classification by Preece and Shinghal [73], there are four main anomaly classes:

- Redundancy;
- Ambivalence;
- Circularity;
- Deficiency.

This classification is depicted in Figure 4.3.

We add more anomalies into this classification and define them in the next section using model theory. Defined anomalies are logical ones in the sense that they do not depend on production rule engine properties such as a rule execution order or priorities. In fact, these anomalies are discovered heuristically and forward chaining of rules is assumed when an anomaly definition is given. This means that in case of verification of other rule types, for instance derivation rules, the same anomalies could be defined differently.

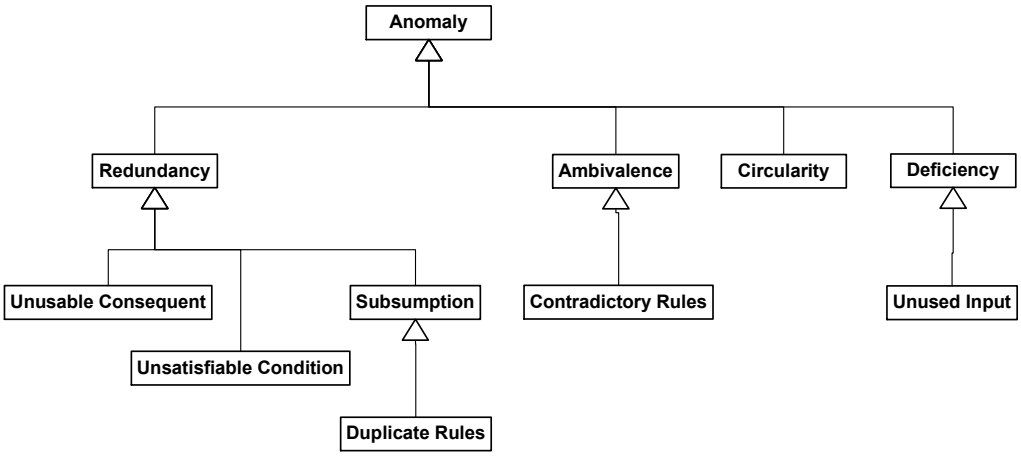

Figure 4.3: Anomaly classification by Preece and Shinghal

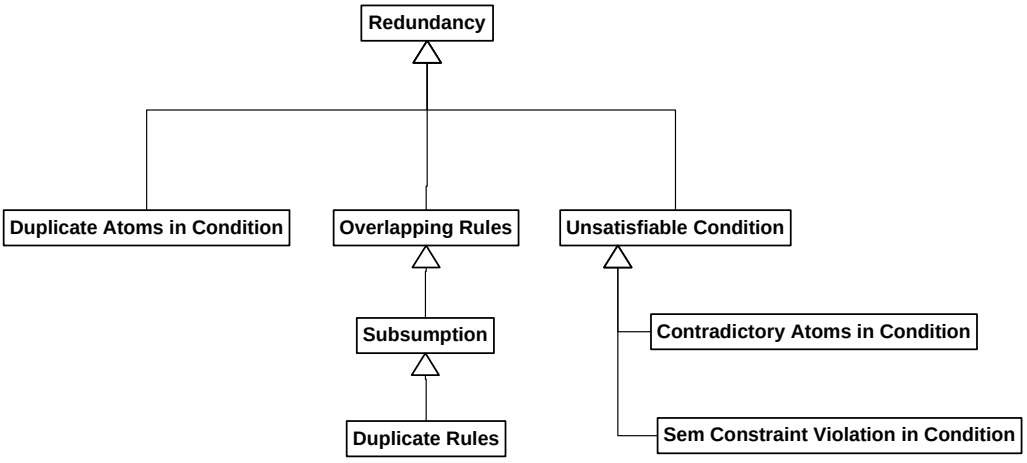

Figure 4.4: Extended classification of redundancy

4.5.1 Redundancy

First, we define different types of redundancy. An extended classification of this anomaly is depicted in Figure 4.4.

Redundancy may occur in two forms: A part of a rule is redundant or a complete rule is redundant. Informally, a rule is redundant if its execution does not affect the state of the fact base, i. e. no facts are added, removed or updated. In some cases, the presence of a redundant rule does not create problems, which may affect expected business results. For instance, the execution of a subsumed rule has the same effect as execution of the rule that subsumes it: Both rules are executed with the same result. The negative effect from such an anomaly is a performance issue and possible difficulties with a rule base maintenance due to increased amount of rules.

In other cases, the appearance of a redundant rule in the rule base is caused by a modeling mistake and it creates errors. For instance, contradictory atoms in the condition of a rule prevent the rule from triggering, which is an error that must be detected and eliminated.

Redundancy: Subsumed Rule

A formula F subsumes another formula F' ($F \succ F'$) if for each interpretation $\mathcal{I}$ if $\mathcal{I} \models \mathcal{F}'$ then $\mathcal{I} \models \mathcal{F}$.

Definition 17 (Subsumed rule)**.** *A production rule r is* subsumed *by another production rule r' if $cond(r')$ is logically implied by $cond(r)$ and $pcond(r)$ is the same as the $pcond(r')$.*

Rule 2 in the running example (Section 4.2) subsumes Rule 1. Let us write these rules in Prolog-like syntax:

Rule 1 Male(?driver)∧ Age(driver, ?x) ∧ ?x< 26 ⇒ YoungDriver(?driver)
Rule 2 Age(?driver, ?x) ∧ ?x< 26 ⇒ YoungDriver(?driver)

Since the right-hand sides of these rules are the same and
Age(?driver, ?x) ∧ ?x< 26 ≻ Male(?driver) ∧ Age(?driver, ?x) ∧ ?x< 26
Rule 2 subsumes Rule 1 by Definition 17.

It is important to notice that the two sample rules above in the Prolog-like syntax are not necessarily anomalous. These rules contain variables, which are considered by the engine as local ones at the rule level. Therefore, even if the variable ?driver is in both rules, it may be unified with different values on the match phase of the execution, resulting in a pair of ground rules without subsumption. In other words, a rule with variables becomes a set of ground rules after possible variables unification, and only some ground rules can be anomalous. However, in this work we perform a syntactic analysis of rules, do not take the content of the fact base into consideration, and do not consider possible unifications. In further examples in this section, we assume, for simplicity, that a variable is local at the rule set level, but verifier rules, which are defined in Section 4.6 for anomaly detection, distinguish between different types of terms (URI, variable, literal) and consider variables at the rule level.

Redundancy: Duplicate Rules

The duplicate rules anomaly is a particular case of a subsumption.

Definition 18 (Duplicate rules)**.** *A rule base contains* duplicate rules *if there are* $r, r' \in R$ *such that they are mutually subsumed.*

Redundancy: Contradictory Atoms in Condition

This anomaly prevents some rules to fire because their conditions cannot be satisfied. Rule 3 in Section 4.2 contains this anomaly:

age(?car,?x) $\wedge$?x$<$ 5$\wedge$?x$>$ 10 $\Rightarrow$premium(?car, 300)

It is obvious that the conjunction ?x$<$ 5$\wedge$?x$>$ 10 cannot be satisfied for all ?x.

Definition 19. *A rule* $r \in R$ *has* contradictory atoms in condition *if does not exist* $\mathcal{I}$*, such that for all variable valuations* $\mathcal{V}$*,* $\mathcal{I}_\mathcal{V} \models cond(r)$*.*

In other words it means that the condition is unsatisfiable if there is no model for the logical formula, which represents the condition.

Redundancy: Semantic Constraint Violation in Condition

First, we define a violation of a semantic constraint by a general logical formula.

Definition 20 (Semantic constraint violation by a logical formula)**.** *A formula* F violates semantic constraint c *if the following holds for all* $\mathcal{I}$*:*

- *If* $\mathcal{I}_\mathcal{V} \models^t F$ *then* $\mathcal{I}_\mathcal{V} \models^f c$

In words, if the formula holds then the constraint does not hold.

Definition 21. *A rule* r *has a* semantic constraint violation in condition *if exists* $c \in C$ *such that* $cond(r)$ *violates* c*.*

Let us consider Rule 8 and Constraint 3 from Section 4.2:

Rule 8: Eligible(?driver) $\wedge\neg$hasTrainingCertification(?driver, 'true') $\Rightarrow$ High-RiskDriver(?driver)
Constr 3: Eligible(?driver) $\wedge$ hasTrainingCertification(?driver, 'true')

It is obvious that if the condition of Rule 8 holds, then Constr 3 does not hold. The business problem, caused by this anomaly is that the rule with such condition is meaningless since existence of facts, on which it holds, is prohibited by the constraint. Or, in other words, the condition of such rule is never satisfied.

Redundancy: Duplicate Atoms in Condition

Duplicate atoms anomaly occurs when two atoms in condition are the same.

Definition 22 (Duplicate atoms in condition)**.** *A rule* $r \in R$ *has* duplicate atoms in condition *if exist* $a_1, a_2 \in cond(r)$ *and* $\mathcal{I}$ *such that for all variable valuations* $\mathcal{V}$ *if* $\mathcal{I}_\mathcal{V} \models^t a_1$ *then* $\mathcal{I}_\mathcal{V} \models^t a_2$*.*

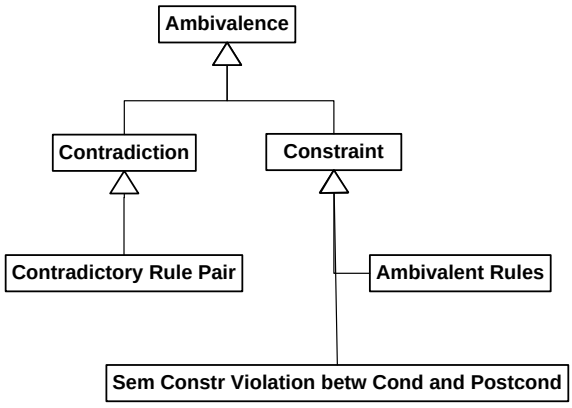

Figure 4.5: Extended classification of ambivalence

4.5.2 Ambivalence

An extended classification of ambivalence types is depicted in Figure 4.5.

Rules are ambivalent if they either violate a semantic constraint or produce contradictory results.

Ambivalence: Contradictory Rule Pair

Two production rules contradict each other in a rule base, if the condition of one of them subsumes the condition of the other, and their postconditions contradict each other in the sense that they are different, for instance, assign different property values. Note, that situations, described by the next two definitions are not contradictions in a logical sense, but points of attention since, depending on the execution order of rules, different results can be asserted.

Definition 23 (Contradictory rule pair)**.** *Two production rules* r_1, r_2 *contradict each other if* $cond(r_1)$ *subsumes* $cond(r_2)$ *and* $pcond(r_1) \neq pcond(r_2)$*.*

This anomaly needs more explanation since it may have different effects, which are not necessary a problem. Let us consider Rule 4 and Rule 5 in Section 4.2:

Rule 4: Luxury(?car) ⇒ basePremium(?car, 500)
Rule 5: Luxury(?car) ∧ age(?car, ?x) ∧ ?x< 5 ⇒ basePremium(?car, 300)

If the property basePremium is single-valued, then these rules contradict each other since the condition of Rule 4 subsumes the condition of Rule 5, but their postconditions are different in the sense that they assign different base premium values. The business problem, caused by this anomaly is that depending on the execution order of rules, different base premium values can be assigned. It might be the case that the property is multi-valued (declared either intentionally or by mistake) like, for instance, in Jena, which uses RDF triples and where properties are multi-valued by default, unless otherwise is explicitly specified in the corresponding ontology. Then such situation is not anomalous, but it still

may be considered as an "attention point": May be the rule modeler forgot to specify the property as single-valued.

We also consider more general case of a contradictory rule pair, when conditions of rules are not subsumed, but may hold at the same time. We refer to such anomaly as a *soft contradiction of rules.*

Let us consider a rule base:

Rule 6 price(?car, ?x) ∧ ?x> 45000 ⇒potentialTheftRating(?car, 'high')
Rule 7 Convertible(?car) ⇒ potentialTheftRating(?car, 'moderate')

If the fact base contains the ground fact *convertible car with the price greater than 45000* then the execution of these rules may lead to different discounts for the car, depending on the order in which rules are executed. In practice a business rule modeler just writes rules without keeping a possible execution order in mind, therefore it is easy to add this anomaly into the rule base. The detection of such anomaly may help in rules debugging and reduces the development time.

Definition 24 (Soft rule pair contradiction)**.** *A rule pair r_1, r_2 has soft contradiction if there is $\mathcal{I}$, such that for all variable valuations $\mathcal{V}$, $\mathcal{I}_\mathcal{V} \models^t cond(r_1) \wedge cond(r_2)$ and $pcond(r_1) \neq pcond(r_2)$.*

Ambivalence: Ambivalent Rule Pair

A rule pair is ambivalent if one rule condition subsumes the other and the conjunction of its postconditions violates a semantic constraint.

Definition 25. *A rule pair $r_1, r_2 \in R$ is* ambivalent *if $cond(r_1)$ subsumes $cond(r_2)$ and there is a semantic constraint $c \in C$ such that the conjunction of postconditions $pcond(r_1) \wedge pcond(r_2)$ violates c.*

Let us consider Rule 9, Rule 10 and the semantic constraint Constr 1 from Section 4.2:

Rule 9 age(?driver, ?x) ∧ ?x> 26 ⇒NormalDriver(?driver)
Rule 10 age(?driver, ?x) ∧ ?x> 70 ⇒SeniorDriver(?driver)
Constr 1 ¬(SeniorDriver(?driver) ∧ NormalDriver(?driver))

The condition of Rule 9 subsumes condition of Rule 10, however, their right-hand sides are different. It means that if these rules are executed, the fact base will contain two facts: The driver is a normal driver and the driver is a senior driver, which is prohibited by the semantic constraint.

Ambivalence: Semantic Constraint Violation by Condition and Postcondition

A production rule has a semantic constraint violation by condition and postcondition if the state of the fact base after execution of the rule violates a semantic constraint.

Definition 26. *A production rule r has a* semantic constraint violation by condition and postcondition *if exists $c \in C$ such that $cond(r) \wedge pcond(r)$ violates c.*

Let us consider Rule 12 and Constraint 2 from Section 4.2:

Rule 12 Provisional(?car) ∧ age(?car, ?x) ∧ ?x> 3 ⇒NotEligible(?car)
Constr 2 ¬(NotEligible(?car) ∧ Provisional(?car))

If Rule 12 is executed then the fact base contains two facts: "a car is provisional" and "a car is not eligible", which is prohibited by the semantic constraint.

4.5.3 Deficiency

In general, a knowledge base contains a deficiency if no rules are fired when it is expected. The possible reason for this anomaly is an *incompleteness*: Either a rule or an atom in the rule is missing. Rule 6 has a condition "car with the price greater than 45000", but the rule base has no rule with the condition "car with the price less than or equal to 45000", which means that no rule is fired when the fact base contains, for instance, "a car X for the price 40000". Such situation means that possibly the rule base is incomplete and more rules have to be added.

Definition 27 (Deficiency)**.** *A knowledge base KB has a deficiency if for some fact $x \in X$ no rule $r \in R$ is fired.*

4.5.4 Other Anomalies

Here we list some anomalies that can be found in the literature, but we do not consider such situations as anomalous due to various reasons.

Auxiliary Rule

Such situation can be anomalous in derivation rule sets. However, in production rule systems, it is quite common to have an auxiliary rule in order to derive intermediate facts, which are used in conditions of other rules. For instance, in our rule-based verification approach, we employ supplementary rules in order to modularize anomaly-detection (verification) rules.

Circularity

Sometimes, the rule execution process becomes infinite. Following our execution model, defined in Section 4.4.4, such a behavior is not possible since the process stops when all rules are executed on corresponding facts once at most. The Jena engine prevents circularity using a mechanism called refraction: Avoidance of adding a rule to the agenda based on already matched patterns. However the following rules are executed in a circle by the Drools engine:

Rule 1 $A \Rightarrow B$ **Rule 2** $B \Rightarrow C$
Rule 3 $\neg D \wedge B \Rightarrow D$ **Rule 4** $C \Rightarrow \neg D$

If we carefully analyze the above rule set, we find out that from the model-theoretic point of view it contains a soft contradiction of rules (Definition 24): If B and C are in the working memory, then the rule set produces different results, depending on the rule execution order.

4.6 Anomaly Detection Using Rule-based Verification Approach

In this section we present verifier rules for detecting anomalies, defined in the previous section. Verifier rules use a generic rule metamodel (Section 4.6.3) as a vocabulary. This metamodel is obtained from the Jena rule metamodel (Section 4.6.1). We formulate verifier rules in plain English and formalize them using the syntax of Drools, which is introduced in Section 4.6.2. Soundness and completeness of verifier rules are discussed in Section 4.6.11.

4.6.1 Jena Rules

Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS, OWL and SPARQL and includes a rule-based inference engine. Main features of Jena 2 are:

RDF API - an API for manipulating an RDF model as a set of RDF triples, integrated parsers and writers for RDF/XML (ARP), N3 and N-TRIPLES, and also has support for typed literals.

ARP - an RDF/XML Parser. Jena 2 version is compliant with RDF Core recommendations. ARP is typically invoked using Jena's read operations, but can also be used standalone.

Persistence - an extension to the Jena Model class that provides persistence for models through the use of a back-end database engine.

Reasoning subsystem - a generic rule based inference engine with configured rule sets for RDFS and for the OWL Lite subset of OWL Full.

Ontology subsystem - an API for working with OWL, DAML+OIL and RDFS.

ARQ - a query engine, which implements both the SPARQL query language and RDQL. SPARQL is an RDF query language and protocol developed within W3C.

Let us consider a rule from the UServ case study [94]:

> If the model of a car has a high theft probability and the price of the car is more than 20000 and less than 45000, then the potential theft rating of the car is high.

This rule in Jena syntax:

```
[AE_PTC04:
1            (?car rdf:type Car)
2            (?car carModel ?carModel)
3            (?carModel highTheftProbability 'false'^^xs:boolean)
4            (?car price ?price)
5            ge(?price,20000)
```

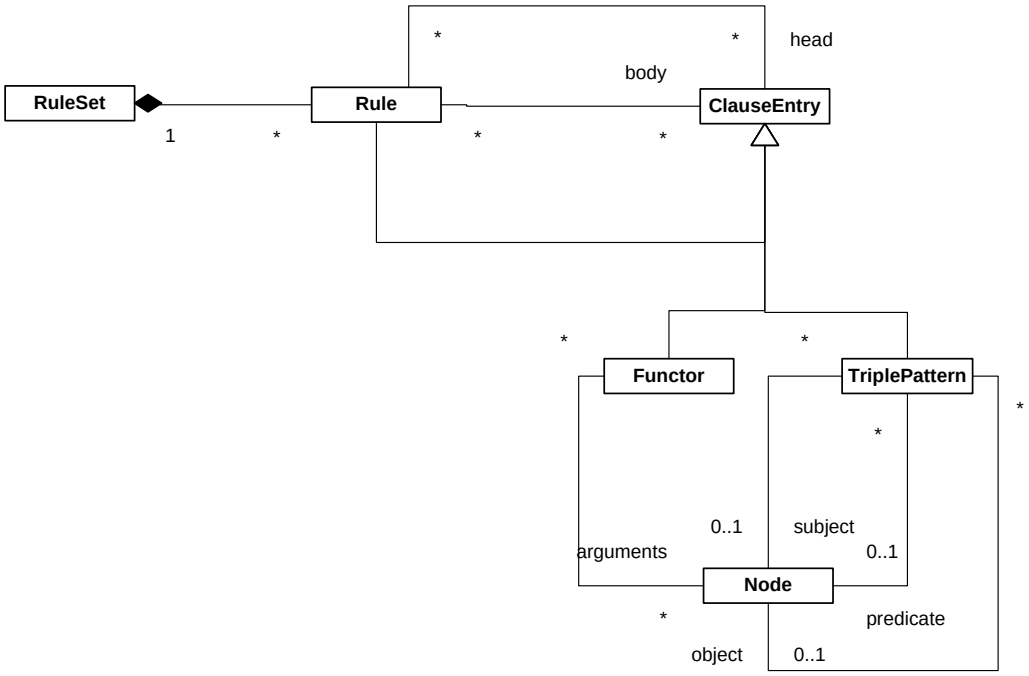

Figure 4.6: Jena rule metamodel

```
6           le(?price,45000)
            ->
7           (?car potentialTheftRating 'moderate')]
```

We introduce the metamodel of Jena rules (see Figure 4.6) and then explain how model elements correspond to the rule elements in the example above.

A Jena rule set consists of rules. Each rule consists of a list of clause entries as a body and a list of clause entries as a head. In the Jena metamodel, "body" is a synonym for "condition" and "head" is a synonym for "postcondition." Further on in the text, we mainly use the words "body" and "head" when speaking about rules in Jena. A ClauseEntry class is a superclass for Rule, Functor and TriplePattern. It means that a rule can be a part of another rule: Jena has a hybrid engine, where a derivation rule can be in the head of a production rule. In our approach, we do not consider such combinations and assume that a head of a production rule may contain only a conjunction of functors and triple patterns.

A Functor class has a name and consists of a list of nodes as arguments. According to the API Jena documentation [3], "*functors play three roles - in heads they represent actions; in bodies they represent built-in predicates; in TriplePatterns they represent embedded structured literals that are used to cache matched subgraphs such as restriction specifications.*".

We consider only those functors which represent built-in predicates.

A TriplePattern class represents a triple of nodes as subject, predicate and object. The Jena Node is depicted in Figure 4.7.

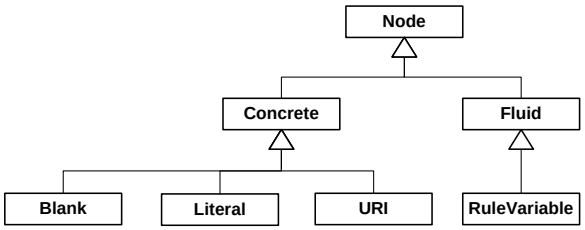

Figure 4.7: Jena rule metamodel

A Node is either a RuleVariable, an URI, a typed Literal, or a Blank node. In the example rule `AE_PTC04` above, lines 1-6 are the body of the rule and line 7 is the head. Lines 1-4 are triple patterns. In the triple `(?car rdf:type Car)` in line 1, `?car` is a rule variable, `rdf:type` is an URI node which denotes the RDF property `type` and `Car` is an URI node, which denotes the class. Lines 5 and 6 contain functors: Greater than or equal (line 5) and less than or equal (line 6). Arguments of these functors are a rule variable (`?car`) and literals (`20000` and `45000`). The triple pattern in the head of the rule (line 7) is asserted into the working memory if all clause entries in the body hold.

4.6.2 JBoss Rules

JBoss Rules (the former Drools rules engine project) [1] provides an open source rule engine that enables easy change and management of rules. Its rule engine implements an enhanced Rete algorithm, named ReteOO (adaptation for an object-oriented interface). As an alternative to the ReteOO algorithm, JBoss Rules offers a hybrid, experimental version of the LEAP algorithm. JBoss Rules is a production rule system. A production rule in JBoss has a two-part structure with a left hand side (the "when" part) and a right hand side (the "then" part). Additionally, a rule may have the following attributes: Salience, agenda-group, auto-focus, activation-group, no-loop, duration. A simple template of a JBoss rule is:

```
rule "rule_name"
    [attribute] [value]
when
    condition
then
    actions
end
```

Let us consider the sample rule, defined in previous Section 4.6.1. This rule in JBoss syntax:

```
rule "AE_PTC04"
when
     $carModel:CarModel(highTheftProbability == false)
     $car:Car(carModel ==  $carModel,
```

```
            price >= 20000,
            price <= 40000
           )
then
     $car.setPotentialTheftRating("moderate");
     modify($car);
end
```

The main concept of a JBoss condition is a Column, for instance:

```
$carModel:CarModel(highTheftProbability == false)
```

It contains zero or more Field Constraints, meaning the Column terms, for instance:

```
carModel ==  $carModel,
price >= 20000,
price <= 40000
```

The entire condition part of a JBoss rule is a tuple of facts (a tuple of Columns). The word Column is used to indicate Field Constraints on a Fact. JBoss Rule Facts from the working memory are bean object instances, so these Field Constraints can be accessed from methods without arguments, also called accessors or getters. When testing the constraint "price <= 25", the bean method getPrice() is used to access the Car instance.

Two Field Constraints are combined with a conjunctive connector, which is the logical conjunction, represented by the comma. So, this rule searches for all the facts in the working memory, which represent a car, with the certain car model and the price between 20000 and 40000. Variables $car and $carModel represent instances of the Car class and the CarModel class. It is a bound variable constraint, named declaration. These instances allow us to access attributes and make function calls of the classes in the "then" (action) part of the rule.

```
$car.setPotentialTheftRating("moderate");
```

The action part of a JBoss rule consists of a block of any valid Java code. Its purpose is either to retract, modify or add facts to the working memory and to invoke specific actions. The previous piece of code sets the potentialTheftRating attribute of a $car to the constant As a consequence of the LHS part of our rule, we set the driverAgeCategory attribute of the value "moderate" by calling the corresponding Java setter.

In order to notify the rule engine of the modified facts, the modify() method is invoked.

```
modify($car);
```

4.6.3 The Generic Rule Metamodel for Jena Rules

In order to verify Jena rules, we use verifier rules that express anomalies using the Jena metamodel as a vocabulary. However, writing verifier rules in terms of the Jena metamodel (Figure 4.6) is a difficult task. This is mainly due to lack of information in the Jena metamodel. For instance, if we need to write a verifier rule that checks for duplicate

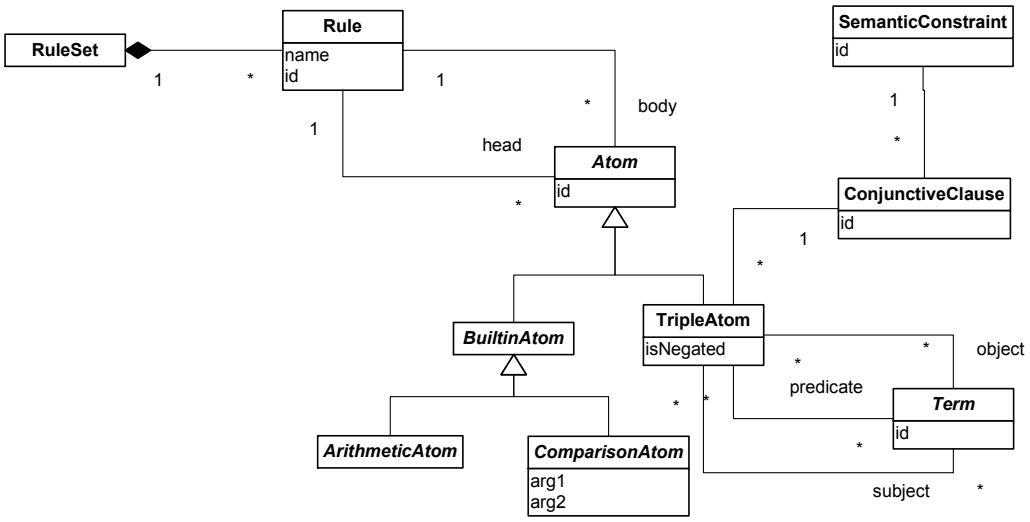

Figure 4.8: The generic rule metamodel

atoms in the condition, we have to query the working memory for two identical atoms and make sure that they belong to the same rule. The latter check is conducted by comparing rule identifiers of the atoms. Jena clause entries (functors and triple patterns) do not have identifiers, therefore verifier rule may become verbose when trying to perform such checks.

In order to simplify development of verifier rules we provide a *generic rule metamodel* for Jena rules. It is very similar to the original Jena metamodel, but it contains some additional information, which is generated automatically by a transformation process called *flattening*. Flattening enriches Jena rules with identifiers and performs some restructuring of Jena functors and triple patterns into simpler objects. For instance, the negation in Jena is implemented by means of noValue(TriplePattern tp) built-in, while in the generic rule metamodel the negation is just a boolean attribute of the class TripleAtom, which simplifies conditions of verifier rules in many cases.

The generic rule metamodel is depicted in Figure 4.8.

A verifier rule set consists of verifier rules. A verifier rule has a name and a unique identifier. It has a list of atoms as a head and a list of atoms as a body. An abstract class Atom has an identifier and an identifier of the rule (ruleId) it belongs to. Class Atom corresponds to the abstract class ClauseEntry in the Jena metamodel. We distinguish two types of atoms: The BuiltinAtom, which represents various built-ins and TripleAtom, which represents Jena triples. The TripleAtom class has the attribute isNegated and refers to the abstract class Term, which represents subject, predicate and object of the triple atom. A BuiltinAtom is either an ArithemticAtom, which represents various Jena built-ins for arithmetic functions, or a ComparisonAtom, which represents Jena comparison built-ins as, for instance, greaterThan or lessThan (Figure 4.9).

We assume that the first argument arg1 in a comparison built-in is a variable, which can always be achieved by an appropriate flattening. For instance, we translate the Jena built-in `ge(3, ?x)` into $?x <= 3$. This assumption reduces the amount of combination cases,

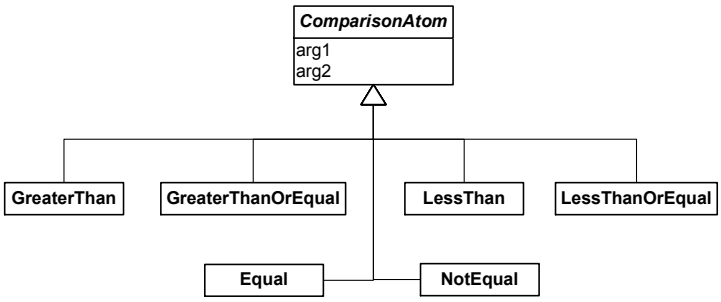

Figure 4.9: Built-in atoms in the generic rule metamodel

which have to be captured by verifier rules and therefore reduces the amount of verifier rules. We represent semantic constraints, which can be defined in the corresponding OWL ontology, as a disjunction of conjunctive clauses. ConjunctiveClause contains a list of triple atoms and refers to a semantic constraint.

4.6.4 Supplementary Rules

Verifier rules are used to detect anomalies in Jena rule sets. Detection of some anomalies may require more than one verifier rule. In order to make verifier rules more compact, we delegate the calculation of some intermediate results to supplementary rules. For instance, some verifier rules have to check whether two atoms are equal in the sense they have the same model. Therefore, we define a supplementary rule that asserts pairs of equal atoms and then we query these pairs from conditions of verifier rules. Supplementary rules do not check for anomalies in rule bases, but assert intermediate data, which is used by verifier rules. Since verifier rules depend on the data produced by supplementary rules, the latter must be executed first.

A comparison of atoms requires comparison of atom constituents, terms:

TermURI : URI references.

TermLiteral : Literals (which consist of a literal value,datatype and language tag).

TermRuleVariable : Variables, which are bound to RDF resource or literal values at run time.

TermAnonymous : A term with an id (name), but not identified by a URI and is not a literal. Normally, it can only be used as a subject or an object of a triple.

Since there are different types of terms, the equality between them has to be defined.

Terms Equality

TermURI : Two URI terms are equal if they have the same URIs.

TermLiteral : Two literal terms are equal if they have the same literal values and the same value types.

TermRuleVariable : Two rule variable terms are equal if they have the same name and belong to the same rule. The second condition is important since rule variables are local at the rule level. Two variables from different rules, but with the same name can be bound with different values.

TermAnonymous : Two anonymous terms are equal if they have the same names.

The equality rules are implemented as equals(Term r) method for each term type.

Terms Inequality

Jena defines datatype built-ins for comparing data term values. In the generic rule metamodel such terms are rule variable terms and literal terms.

A literal term can be compared to either another literal term or a rule variable. If a literal term is compared to another literal term of the same datatype, then a comparison is conducted according to the order, defined for the datatype. The possible results are "less than", "greater than", "less than or equal", "greater than or equal" and "equal".

If a literal term is compared to a rule variable, then the result is "unknown". This is because the rule variable is bound when the rule is executed and its value is not known during the verification process.

If a rule variable is compared against another rule variable then the result is either "unknown" or "equal" if variables have the same name and belong to the same rule.

The inequality rules are implemented as a compareValues(Term t) method for TermLiteral and TermRuleVariable classes respectively.

Duplicate Atoms

Duplicate atoms are queried, for instance, in the verifier rule for subsumption (Section 4.6.6).

Supplementary Rule 1 (Duplicate triple atoms). Two triple atoms are *equal* if their subjects, predicates, and objects are equal and $\models^t a_1 \longrightarrow \models^t a_1 \wedge a_2$.

This supplementary rule detects duplicate atoms across the whole rule set.

```
rule "Redundant (duplicate) triple atoms"
    when
1       $left :TripleAtom()

2       $right :TripleAtom(
3           id != $left.id,
4           negated == $left.negated
5       )

       #Check that subject, predicate, object are the same
6       eval($left.getSubject().equals($right.getSubject()))
7       eval($left.getPredicate().equals($right.getPredicate()))
8       eval($left.getObject().equals($right.getObject()))


    then
```

```
9        insert(new DuplicateAtom( $left, $right ));
end
```

For certain purposes, for instance, while detecting semantic constraints violation, we query for pairs of oppositely negated atoms.

Supplementary Rule 2 (Oppositely negated triple atoms). A triple atom a_1 is *oppositely negated* to triple atom a_2 if their subjects, predicates, and objects are equal and $\models^f a_1 \wedge \neg a_2$.

This supplementary rule detects pair of oppositely negated atoms across the whole rule set.

```
rule "Redundant (duplicate) triple atoms"
    when
1        $left :TripleAtom()

2        $right :TripleAtom(
3            id != $left.id,
4            negated != $left.negated
5        )

        #Check that subject, predicate, object are the same
6        eval($left.getSubject().equals($right.getSubject()))
7        eval($left.getPredicate().equals($right.getPredicate()))
8        eval($left.getObject().equals($right.getObject()))


    then
9        insert(new OppositelyNegatedAtoms( $left, $right ));
end
```

It is possible to have not only duplicate triple atoms, but also duplicate built-in atoms. For each built-in atom a corresponding supplementary rule has to be defined. For instance, the following supplementary rule derives duplicated BuiltinGreaterThan atoms.

Supplementary Rule 3 (Duplicate greater than built-in atoms). Two BuiltinGreaterThan atoms are *equal* if they first and second arguments are correspondingly equal.

```
rule "Dduplicate builtin greater than atoms"
    when
        $left :BuiltinGreaterThan()

        $right :BuiltinGreaterThan(
            id != $left.id,
            arg1 == $left.arg1,
            arg2 == $left.arg2
        )
    then
        insert(new DuplicateAtom( $left, $right ));
end
```

Contradictory Atoms

Pairs of contradictory atoms are used, for instance, in the verifier rule for detection of contradictory rule pairs (Section 4.6.8).

Supplementary Rule 4 (Contradictory triple atoms). Two triple atoms are *contradictory* if they are in the head of the rule, their subjects and predicates are equal, but objects are not equal.

In this rule "equal" and "not equal" have the meaning defined above for equality and inequality of terms.

```
rule "Contradictory triples"
    when
        $left :TripleAtom(body == false)

        $right :TripleAtom(
            ruleId != $left.ruleId,
            id != $left.id,
            negated == $left.negated,
            body == false
        )

        #Check that subject and predicate are the same, objects are different
        eval($left.getSubject().equals($right.getSubject()))
        eval($left.getPredicate().equals($right.getPredicate()))
        eval(!$left.getObject().equals($right.getObject()))

    then
        insert( new ContradictoryTriples( $left, $right ));
end
```

4.6.5 Redundancy: Contradictory Atoms in Condition

This anomaly in Jena rules is caused either by oppositely negated atoms or by opposite number and date/time ranges.

Opposite Negation of Triples in Condition

Let us consider the following Jena rule:

```
[Oppositely negated triples:
1       (?car eg:color 'red')
2       noValue(?car eg:color 'red')
        ->
       //head
]
```

It contains oppositely negated triples in the body: A triple in line 1 is negated in line 2 by means of Jena noValue() built-in.

Verifier Rule 1 (Opposite negation of triples in condition). Two triple atoms have *opposite negation* if they are from the same rule body, and they have the same subject, predicate and object, but they are oppositely negated.

The rule in Drools syntax:

```
rule "Opposite negation"
    when
        $left :TripleAtom()
        $right :TripleAtom(
            #Check that these atoms are from the same rule
            ruleId == $left.ruleId,

            #Check that they have different identifiers
            id != $left.id,

            #Check that they are negated oppositely
            negated != $left.negated
        )

        #Check that subject, predicate, object are the same
        eval($left.getSubject().equals($right.getSubject()))
        eval($left.getPredicate().equals($right.getPredicate()))
        eval($left.getObject().equals($right.getObject()))

        # Check that there is not already a pair with these values
        not Opposites(
            left == $right,
            right == $left
        )
    then
        insert( new Opposites( $left, $right ));
end
```

Opposite Number and Date Ranges

Jena has special built-ins for comparing integers, reals and dates. In the generic rule metamodel (Section 4.6.3) these built-ins are represented by the following classes: BuiltinGreaterThan, BuiltinLessThan, BuiltinGreaterThanOrEqual, BuiltinLessThanOrEqual, BuiltinEqual, BuiltinNotEqual. These classes correspond to built-in atoms in the generic rule metamodel (Section 4.6.3).

Let us consider the following Jena rule:

```
[Opposite integer ranges:
       ge(?price,10)
       lessThan(?price,10)
        ->
       //head
]
```

It is obvious that built-in atoms in the body of the rule are in contradiction and therefore the rule cannot be fired.

Verifier Rule 2 (Opposite integer ranges). Let var be a variable and val_1, val_2 be integers. A built-in atom GreaterThanOrEqual(var, val_1) and a built-in atom LessThan(var, val_2) from the same rule body have opposite integer ranges if val_1 is greater than or equal than val_2.

The rule in Drools syntax:

```
rule "Opposite Integer ranges (greaterThanOrEqual lessThan)"
    when
        $left :BuiltinGreaterThanOrEqual()

        $right :BuiltinLessThan(
            ruleId == $left.ruleId,
            id != $left.id
        )
        #Checks that first arguments are the same
        eval($left.getArg1().equals($right.getArg1()))

        #Checks that arg2 of $left is greater than or equal to arg2 of $right
        (
          eval($left.getArg2().compareValues($right.getArg2())==
                                                    BuiltinCompareType.GREATER_THAN)
           or
          eval($left.getArg2().compareValues($right.getArg2())==BuiltinCompareType.EQUAL)
        )

        # Check that there is not already a pair with these values.
        not Opposites(
            right == $left,
            left == $right
        )
    then
        insert( new Opposites( $left, $right ));
end
```

There are several anomalous combinations of comparison built-ins. If a variable is compared against another variable, then the situation is not necessary anomalous. For instance, whether the condition of the following rule has the opposite range anomaly depends on values of `?y` and `?z`:

```
[rule3:
       ge(?price,?y)
       le(?price,?z)
        ->
       //head
]
```

In this case the verifier rule asserts a warning, but not an error.

4.6.6 Redundancy: Subsumed Rules and Duplicate Rules

Rule 2 from the running example subsumes Rule 2 since the body of Rule 2 subsumes the body of Rule 1 and their heads are the same (see rules in Jena syntax in Section A.1).

Verifier Rule 3 (Subsumed rule). Rule r_1 *subsumes* r_2 if:

- for each atom t_1 in the body of r_1 there is an atom t_2 in the body of r_2 such that t_1 is equal to t_2 (see Supplementary Rule 1 for atoms equality);
- the head of r_1 is equal to the head of r_2.

The rule in Drools syntax:

```
rule "Subsumed rules"
    when
        $r1 :Rule()
        $r2 :Rule(
            id != $r1.id
        )

        # Check that the body of r1 subsumes the body of r2
        forall(
           $atom: Atom(ruleId == $r1.id,
                              body == true)
           DuplicateAtom(
                left == $atom,
                right memberOf $r2.body
           )
        )

        # Next two 'forall' check that heads of r1 and r2
        # are equal (mutually redundant, equal)
        forall(
           $atom: Atom(ruleId == $r1.id,
                           body == false)
           DuplicateAtom(
                left == $atom,
                right memberOf $r2.head
           )
        )
        forall(
           $atom: Atom(ruleId == $r2.id,
                           body == false)
           DuplicateAtom(
                left == $atom,
                right memberOf $r1.head
           )
        )
    then
        insert( new Subsumption( $r1, $r2 ));
end
```

Before this verifier rule is executed, equal atoms have to be asserted into the working memory using the supplementary rule in Section 4.6.4 (duplicate atoms). In order to detect duplicate rules, the rule above has to be modified with the additional check for the mutual subsumption of bodies.

4.6.7 Redundancy: Duplicate Atoms in Condition

A verifier rule, which detects duplicate atoms in rule conditions is similar to the supplementary rule for duplicate atoms in a rule base (Section 4.6.4). An additional condition that requires `$left` and `$right` to be in the same rule body is added:

```
        $right :TripleAtom(
            id != $left.id,
            negated == $left.negated,
            # this atom is from the same rule
            ruleId == $left.ruleId,
```

```
            # both atoms are from the body
            body == true
        )
```

4.6.8 Ambivalence: Contradictory Rule Pairs

Verifier Rule 4 (Contradictory rule pair). Rule r_1 *contradicts* rule r_2 if the following holds:

- the body of r_1 subsumes the body of r_2;
- for each triple t_1 in the head of r_1 there is a contradictory triple t_2 in the head of r_2;
- for each triple t_1 in the head of r_2 there is a contradictory triple t_2 in the head of r_1.

See Rule 4 and Rule 5 from Section 4.2 in Jena syntax in Section A.1. This verifier rule employs the data, asserted by the supplementary rule, defined in Section 4.6.4: two 'forall' operators in the rule below check that rule heads contradict each other.

```
rule "Contradictory rule pairs"
    when
        $r1 :Rule()
        $r2 :Rule(
            id != $r1.id
        )

        # Check that the body of r1 subsumes the body of r2
        forall(
           $triple: TripleAtom(ruleId == $r1.id,
                               body == true)
           DuplicateAtom(
                left == $triple,
                right memberOf $r2.body
           )
        )

        # Next two 'forall' check that heads of r1 and r2 are contradictory
        forall(
           $triple: TripleAtom(ruleId == $r1.id,
                               body == false)

           ContradictoryTriple(
                left == $triple,
                right memberOf $r2.head
           )
        )
        forall(
           $triple: TripleAtom(ruleId == $r2.id,
                               body == false)

           ContradictoryTriple(
                left == $triple,
                right memberOf $r1.head
           )
        )
```

```
    then
        insert( new ContradictoryRulePair( $r1, $r2 ));
end
```

4.6.9 Deficiency: Missing Atoms

One possibility for a deficiency as defined by Definition 27 is a missing atom, which is one of the following in Jena:

- Incomplete number ranges. For instance, when a body of a rule contains an atom $a > b$, then for completeness some other rule body should have the atom $a <= b$;
- Missing oppositely negated atom. For instance, if a body of a rule has a triple, then some other rule body should contain the negation of the triple. In practice, such case is rarely anomalous;
- Missing equality atom. For instance, Rule 13 has an atom "the potential theft rating of a car is 'high'", where "potential theft rating" is a property, which may have one of three values: 'low', 'medium', and 'high'. The sample rule set has a deficiency since it does not contain rules with equality atoms, which check the property "potential theft rating" for values 'low' and 'medium'.

Let us define a verifier rule for detecting incomplete integer ranges. For each comparison built-in atom (greater than, greater than or equal, less than, less than or equal) we define a verifier rule.

Verifier Rule 5 (Incomplete integer range (greater than)). A rule set has an *incomplete integer range* if there is a built-in "greater than" atom $A := (a_1 > a_2)$ such that none of the following holds:

- There is a built-in "less than or equal" atom $a_1 <= a_2$ which is not in the condition of the same rule as A.
- There is a built-in "equality" atom $a_1 = a_2$ and built-in "less than" atom $a_1 < a_2$ which are not in the condition of the same rule as A.
- There is a built-in "less than" atom $a_1 < a_2$ which is not in the condition of the same rule as A.

```
rule "Incomplete integer range (greater than)"
    when
     $t1 :BuiltinGreaterThan()
     not(
        exist(
           BuiltinLessThanOrEqual(ruleId != $t1.ruleId,
                                  a2 >= $t1.a2
                                 )
        )
        exists(
           BuiltinEqualityAtom(ruleId != $t1.ruleId,
                               a2 == $t1.a2
                              ) and
```

```
            BuiltinLessThan(ruleId != $t1.ruleId,
                            a2 == $t1.a2
                           )
        )
        exists(
            BuiltinLessThan(ruleId != $t1.ruleId,
                            a2 >= $t1.a2
                           )
        )
     )
    then
        insert(new IncompleteIntegerRange($t1.getRule()));
end
```

In order to cover all possible combinations of incomplete integer ranges we need four rules (each for every comparison built-in).

4.6.10 Semantic Constraints Violation

In this section we define verifier rules for three anomalies: semantic constraint violation in condition, ambivalent rule pair and semantic constraint violation by condition and postcondition.

A semantic constraint is a logical formula, which describes an inconsistent state. This concept is similar to constraints in relational databases where they are interpreted as checks during database updates.

In Jena, semantic constraints may come from a corresponding OWL DL ontology. An important point here is that we interpret the schema part of an OWL DL ontology (called *TBox*) as integrity constraints as it is described in [61]. Such interpretation is different from the original meaning of OWL axioms in its effect. In order to explain the difference, let us consider a sample constraint, expressed by means of the following logical formula:

$$\forall x[\text{Wine}(x) \rightarrow \exists y : \text{producer}(x, y) \wedge \text{Winery}(y)] \tag{4.1}$$

It is equivalent to the following OWL DL axiom:

Class(Wine, restriction(producer, someValueFrom(Winery)))

From the fact Wine(Merlot) and the axiom we can conclude that Merlot has some unknown producer. Hence, the ontology is satisfiable. However, if a database schema contains the sample constraint 4.1, then the assertion of the fact Wine(Merlot) violates the constraint since the producer of the wine is not specified.

In our verification approach we consider an OWL DL ontology as a set of integrity constraints, expressed by logical formulae. This simplifies the task of writing verifier rules because a logical formula has the structure, similar to conditions of Jena rules. Issues of translating from Description Logic into predicate logic are described in various books and articles ([17], [64], [60], [87]). For instance, in [17] the expressive power of DL is compared against the expressive power of predicate calculus. In particular, it describes the translation from descriptions to predicate calculus, which introduces new variables whenever a new quantifier appears.

An approach to ontology modeling by means of Description Logic Programs (DLP) is presented in [87], which explains how DLP can be used for modeling a Horn fragment of OWL DL.

Following [62], "OWL ontologies can be understood as incomplete databases", however in many business applications, which employ relational databases, knowledge is considered complete and integrity constraints control whether all necessary information is provided explicitly. We consider the consistency issue as important and our motivation for interpreting DL axioms as integrity constraints is to allow rule modelers to maintain consistency of the data, asserted by rules. In further discussion under semantic constraints we mean integrity constraints, obtained from the corresponding OWL DL ontology.

Let us consider several examples of Constr 1 from Section 4.2: *No driver is a normal driver and a senior driver*. It can be formalized by means of OWL DL disjointness axiom:

DisjointClasses(NormalDriver, SeniorDriver)

This DL axiom corresponds to the following predicate logic formula:

$$\forall x(\neg\ (\text{NormalDriver}(x) \wedge \text{SeniorDriver}(x)))$$

Another example is Constr 3: *Every eligible driver must have a training certificate*. It is formalized by means of the following OWL DL axiom:

Class(EligibleDriver, restriction(hasCert, someValuesFrom(TrainingCert)))

This DL axiom corresponds to the following predicate logic formula:

$$\forall x(\text{EligibleDriver}(x) \rightarrow \exists y|\text{hasCert}(x, y) \wedge \text{TrainingCert}(y))$$

This implication is equivalent to the following formula in disjunctive normal form:

$$\forall x(\neg\text{EligibleDriver}(x) \vee \neg(\exists y|\text{hasCert}(x, y) \wedge \text{TrainingCert}(y)))$$

Therefore, we represent an OWL DL axiom by means of a logical formula in the disjunctive normal form. In order to check whether such semantic constraint is violated by a conjunction of atoms by means of verifier rules, we have to check that every conjunction of the disjunction is violated (a disjunction of logical formulae is false if each formula is false).

Redundancy: Semantic Constraint Violation in Condition

The condition of Rule 8 from Section 4.2 violates Constraint 3.

A semantic constraint is a logical formula in the disjunctive normal form: $(A_{1,1} \wedge ... \wedge A_{1,n}) \vee ... \vee (A_{n,1} \wedge ... \wedge A_{n,n})$. Therefore, in order to violate such constraint, the condition of a rule must violate every conjunctive clause of the constraint, or, in other words, if the condition of the rule holds, then the semantic constraint does not hold. We use Supplementary Rule 2, which derives pairs of oppositely negated atoms in order to check for conjunctive clauses violation.

Verifier Rule 6 (Sem. constr. violation in condition). A rule r has a semantic constraint violation sc in condition, if $cond(r)$ violates every conjunctive clause $cc \in sc$. Condition $cond(r)$ violates conjunctive clause cc if exists atom $a_1 \in cc$ such that it is oppositely negated with some atom $a_2 \in cond(r)$.

This verifier rule in Drools syntax:

```
rule "Semantic constr violation in condition"
    when
        $sc :SemanticConstraint()

        $r :Rule()

        # Check that every conjunctive clause has an oppositely negated triple
        # in the rule body
        forall(
           $clause:ConjunctiveClause(constrId == $sc.id)

           exists(
              $triple:TripleAtom(clauseId == $clause.id)
              OppositelyNegatedAtoms(
                left == $triple,
                right memberOf $r.body
              )
           )
        )

    then
        insert( new SCVInCond( $r, $sc ));
end
```

Ambivalence: Ambivalent Rule Pair

Rule 9 and Rule 10 from Section 4.2 violate Constraint 1 since rules may derive a fact that a driver is a normal driver and a senior driver at the same time, which is prohibited by the constraint.

Verifier Rule 7 (Ambivalent rule pair). A rule pair r_1, r_2 is ambivalent if there is a semantic constraint c such that:

1. cond(r_1) subsumes cond(r_2).
2. Every conjunctive clause of c contains an atom, which is oppositely negated to some atom a either from pcond(r_1) or from pcond(r_2);
3. In pcond(r_1) exists an atom, which is in c;
4. In pcond(r_2) exists an atom, which is in c.

Condition 1 is important to make sure that rules can be executed on the same facts. Condition 2 in this rule checks whether constraint c is violated by either atoms from the head of rule 1 or from the head of rule 1. However, this does not guaranty that the conjunction of rule heads violates c, since it is possible that the constraint is violated by

the head of just one rule. For instance, if the head of the rule is $A \wedge B$ and constraint is $\neg A \wedge \neg B$ then the constraint is violated no matter of the head of the second rule. Additional conditions 3 and 4 guarantee that each rule head contains at least one oppositely negated atom from c and, therefore, a conjunction of rule heads violates the constraint.

```
rule "Ambivalent rule pair"
    when
        $sc :SemanticConstraint()

        $r1 :Rule()
        $r2 :Rule(id != r1.id)

        # Check that the body of r1 subsumes the body of r2
        forall(
           $atom: Atom(ruleId == $r1.id,
                       body == true)
           DuplicateAtom(
                left == $atom,
                right memberOf $r2.body
           )
        )

        # Check that every conjunctive clause is violated
        # by conjunction of pcond(r1) and pcond(r2)
        forall(
           $clause:ConjunctiveClause(constrId == $sc.id)

           exists(
              $triple:TripleAtom(clauseId == $clause.id)
              or(
                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r1.head
                   )

                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r2.head
                   )
              )
           )
        )


        # At least one atom from the head of r1 and r2
        # must be in some conjunctive clause of sc
        exists(
           $a1:Atom(ruleId == $r1.id,
                    body == false
                   )
           OppositelyNegatedAtoms(
                 left memberOf $sc,
                 right == $a1
           )
        )
```

```
        exists(
           $a1:Atom(ruleId == $r2.id,
                    body == false
                   )
           OppositelyNegatedAtoms(
                    left memberOf $sc,
                    right == $a1
           )
        )
    then
        insert( new AmbivalentRulePair( $r1, $r2, $sc ));
end
```

Ambivalence: Semantic Constraint Violation by Condition and Postcondition

Condition and postcondition of Rule 12 from Section 4.2 violate Constraint 2 since Rule 12 can derive a fact that some car is provisional and not eligible at the same time, which is prohibited by the constraint.

Verifier Rule 8 (Sem. constr. violation by cond. and postcond.). A rule r violates semantic constraint c by condition and postcondition if

1. Each atom of every conjunctive clause of c has an oppositely negated atom either in $\text{cond}(r)$ or in $\text{pcond}(r)$;
2. At least one atom of $\text{pcond}(r)$ is in c.
3. At least one atom of $\text{cond}(r)$ is in c.

First condition checks whether condition or postcondition of the rule violates every conjunctive clause of the constraint, abd therefor, violates the constraint. Conditions 2 and 3 check that the conjunction of condition and postcondition violates the constraint. This verifier rule in Drools syntax:

```
rule "semantic constraint violation by condition and postcondition"
    when
        $sc :SemanticConstraint()

        $r :Rule()

        forall(
           $clause:ConjunctiveClause(constrId == $sc.id)
           exists(
              $triple:TripleAtom(clauseId == $clause.id)
              or(
                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r.body
                   )

                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r.head
                   )
              )
```

```
            )
        )

        # At least one atom from the the body of r and the head of r2
        # must be in some conjunctive clause of sc
        exists(
           $a:Atom(ruleId == $r.id,
                    body == false
                   )
           OppositelyNegatedAtoms(
                 left memberOf $sc,
                 right == $a
           )
        )
        exists(
           $a1:Atom(ruleId == $r.id,
                    body == true
                   )
           OppositelyNegatedAtoms(
                    left memberOf $sc,
                    right == $a
           )
        )
    then
        insert( new SemcCVByCondPcond( $r, $sc ));
end
```

4.6.11 Soundness and Completeness of the Rule-based Verification Approach

The discussion concerning soundness and completeness of the proposed verification solution is about relations between model-theoretic anomaly definitions (Section 4.5) and verifier rules, which detect defined anomalies (Section 4.6). Informally, a solution is sound if it solves the problem for which it is developed. In our case, the solution is a set of verifier rules for the detection of different anomalies. The converse of the soundness property is the completeness property. A solution is complete if its set of verifier rules detects all possible anomalies.

Soundness

Let us define the soundness of Verifier Rule 1, which detects **opposite negation of triples in condition**.

Let $A(a_1, a_2)$ be an anomaly, detected by Verifier Rule 1 and a_1, a_2 are triple atoms. We say that this rule is *sound* if does not exist $\mathcal{I}$, such that $\mathcal{I} \models a_2 \wedge a_2$.

Since the verifier rule searches for oppositely negated atoms by definition, it is obvious that the conjunction of a_1 and a_2 cannot be satisfied. Therefore, the verifier rule is sound.

The soundness of verifier rules for detection of other anomalies can be shown following the same two-steps principle:

1. The soundness of verifier rules for detection of a specific anomaly is defined;

2. A pair of rules, asserted by the verifier rules is analyzed and it is shown that the pair fits the model-theoretic definition of the anomaly, given in Section 4.5.1, and, therefore, the verifier rules are sound according to the definition, given in the step 1.

Let us follow these two steps in order to show the soundness of the verifier rule for the subsumption anomaly:

1. Let $S(r_1, r_2)$ be a subsumption anomaly, detected by Verifier Rule 3, where rule r_1 is subsumed by rule r_2. We say that this verifier rule is *sound* if $cond(r_2)$ is logically implied by $cond(r_1)$ and $pcond(r_1)$ is the same as $pcond(r_2)$.

2. Let us analyze whether rules r_1 and r_2 fit the definition in step 1. Since the verifier rule checks that every atom $a \in cond(r_2)$ is in $cond(r_1)$, then $cond(r_2)$ is logically implied by $cond(r_1)$.

   It is obvious from the verifier rule, which detects the anomaly, that postconditions of r_1 and r_2 are the same (equal). Therefore, Verifier Rule 3 is sound by the definition, given in step 1.

Completeness

As it is stated in [59] and further supported in the tutorial and survey on rule verification by O'Keefe [66], "verification, based upon anomaly detection is a heuristic approach, rather than deterministic, for two reasons. First, detected anomalies may not be errors, and errors may exist that are not related to known anomalies. Second, some of the methods for detecting anomalies are themselves heuristic, and thus do not guarantee detection of all identifiable anomalies".

Since the rule-based verification approach presented in this work is heuristic in the sense that verifier rules are created using case-studies and common sense, we do not vouch for its completeness. However, if an anomaly which is not yet covered by the verifier rules is discovered, the completeness of the approach can easily be increased by adding new verifier rules.

4.7 Limitations and Conclusions

In this chapter, we have presented a declarative approach to rule verification. The main idea is to detect anomalies by means of verifier rules. In our research we verify business rules expressed in Jena and we use JBoss Rules in order to express and execute verifier rules. The achieved verification results can be useful for rule application developers, who, in particular, use Jena for building Semantic Web applications. In fact, the verification approach can easily be extended to the verification of rules, expressed in any rule language. Similar work has been conducted for verification of JBoss rules [79].

The main advantage of the presented work is flexibility: A set of verifier rules can easily be extended if new anomalies are discovered and need to be detected.

As a possible drawback of the approach is a large amount of verifier rules in some cases, which may lead to maintenance problems. For anomalies we have discussed in this work,

the total amount of verifier rules is less than 100, which is not too much. Moreover, verifier rules are packaged by anomalies they detect. This means that they can be debugged and executed selectively, which simplifies maintenance.

Another possible problem is the completeness of the verifier rule set. As we have discussed in the previous section, the approach does not guarantee the full coverage for anomalies. At this moment we leave the coverage issue for the further research. Other possible improvements of the presented approach are related to the universal metamodel for rule verification, expressed in different rule languages. Since these improvements have common ground with the rule interchange approach, we discuss them in the thesis conclusion chapter (Chapter 5).

Chapter 5

Conclusions

We conclude the thesis with a discussion about relations between two main chapters: Chapter 3 on rule interchange and Chapter 4 on rule verification.

Possible future research on rule-based verification is related to the design and practical evaluation of the general rule metamodel, which can be used for verifying rules expressed in various rule languages. Rules are translated into the general rule metamodel, which allows a universal verifier for different rule languages. A good candidate for such metamodel is the R2ML metamodel. R2ML is initially designed for the rule interchange and can be used for encoding Jena rules, JBoss rules, SWRL, OCL and many others. There are several translators implemented, which translate rules from different languages into R2ML.

However, the issue of using the R2ML metamodel as a vocabulary for verifier rules was not the R2ML design goal. It means that the R2ML metamodel may need some extensions. Another R2ML feature which may complicate verifier rules and make them more verbose, is the R2ML rich syntax. This feature is a strong advantage of R2ML as an interchange language since it allows loss-free interchange between very distinct rule languages. On the other hand, the variety of atom types in R2ML may require a lot of verifier rules, which perform anomaly checks for all possible atom types.

There are a number of ideas, which may employ research results on the rule interchange, but which have not been carefully investigated yet. Since during an interchange process rules are translated into a rule interchange format, a lot of services can be provided on top of this intermediate representation. Such services are rule-platform independent and can solve different rule-related tasks. For instance, a verbalization of the general rule markup language R2ML [51] can be used for any rule language for which exists an interchange translator. A platform independent rule verification can also be performed on top of an intermediate rule representation.

Currently, it is hard to make predictions about general further developments in the area of rules, especially rules for the Semantic Web. However, recent initiatives in OMG and W3C concerning standardization and interchange of production rules are inspired by interoperability needs within the business rules community and among vendors of production rule systems. This interest, in turn, motivates us to think one step further and to focus on such upcoming issues as the problem of rule interchange correctness, which is not yet considered neither by the OMG nor by W3C, and the problem of rule verification, which becomes painful as the use of rules in Semantic Web applications grows.

The conducted research on rule interchange, presented in the thesis and the declarative verification of Semantic Web rules, expressed in Jena, can be considered as first solutions to these problems and as a basis for further works on rules interoperability and verification.

Appendix A

Sample Rules

A.1 Rules in Jena Syntax

Rule 1: If the driver is male and is under the age of 26, then he is a young driver.
Rule 2: If the driver is under the age of 26, then he is a young driver.

```
[rule1:
       (?driver rdf:type userv:Male)
       (?driver userv:age ?age)
       lessThan(?age, 26)
        ->
       (?driver rdf:type userv:YoungDriver)
]

[rule2:
       (?driver userv:age ?age)
       lessThan(?age, 26)
        ->
       (?driver rdf:type userv:YoungDriver)
]
```

Rule 3: If a car is the luxury car, then base premium is 500.
Rule 5: If the car is less than 5 years old and is luxury car, then base premium is 300.

```
[rule4:
       (?car rdf:type userv:LuxuryCar)
       ->
       (?car userv:basePremium '500')]
]

[rule5:
       (?car rdf:type userv:LuxuryCar)
```

```
        (?car userv:age ?age)
        lessThan(?age, 5)
        ->
        (?car userv:basePremium '300')]
]
```

Rule 8: If the driver is eligible and has no training certification, then he is a high risk driver.

```
[rule8:
        (?driver rdf:type userv:Eligible)
        noValue((?driver userv:hasTrainingCert 'true'))
        ->
        (?driver rdf:type userv:HighRiskDriver)
]
```

Rule 9: If the driver is over the age of 26, then he is a normal driver.
Rule 10: If the driver is over the age of 70, then he is a senior driver.

```
[rule9:
        (?driver userv:age ?age)
        greaterThan(?age, 26)
        ->
        (?driver rdf:type userv:NormalDriver)
]

[rule10:
        (?driver userv:age ?age)
        greaterThan(?age, 70)
        ->
        (?driver rdf:type userv:SeniorDriver)
]
```

Rule 12: If the car is provisional and is older than 3 years, then it is not eligible car.

```
[rule12:
        (?car rdf:type userv:Provisional)
        (?car userv:age ?age)
        greaterThan(?age, 3)
        ->
        (?car rdf:type userv:NotEligible)
]
```

A.2 Constraints in OWL Abstract Syntax and as Logical Formulae

Constr 1: No driver is a normal driver and a senior driver.
OWL abstract syntax

DisjointClasses(NormalDriver, SeniorDriver)

As a predicate logic formula:

$$\forall x(\neg\ (\text{NormalDriver}(x) \wedge \text{SeniorDriver}(x)))$$

Constr 2: No car eligibility is "provisional" and "not eligible".
OWL Abstract Syntax

DisjointClasses(NotEligibleCar, ProvisionalCar)

As a predicate logic formula:

$$\forall x(\neg\ (\text{NotEligibleCar}(x) \wedge \text{ProvisionalCar}(x)))$$

Constr 3: Every eligible driver has training certification.
OWL Abstract Syntax

Class(EligibleDriver, restr(hasTrainingCert, someValuesFrom(TrainingCert)))

As a predicate logic formula:

$$\forall x\ (\text{EligibleDriver}(x) \rightarrow \exists y\ \text{hasTrainingCert}(x, y) \wedge \text{TrainingCert}(y))$$

Bibliography

[1] Business Rules Management System Drools. Project homepage. http://www.jboss.org/drools.

[2] DOM Model. W3C Recommendation. http://www.w3.org/DOM/.

[3] Jena - A Semantic Web Framework for Java. Project homepage. http://jena.sourceforge.net.

[4] Object Constraint Language (OCL), v2.0. OMG Final Adopted Specification.

[5] OWL Web Ontology Language Semantic and Abstract Syntax. W3C Recommendation 10 February 2004. http://www.w3.org/2004/OWL.

[6] Production Rule Representation (PRR). OMG Adopted Specification. http://www.omg.org/spec/PRR/1.0/.

[7] Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-concepts/.

[8] SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission 21 May 2004. http://www.w3.org/Submission/SWRL/.

[9] ILog JRules 4.6. Technical white paper, 2004.

[10] Ritu Agarwal and Mohan Tanniru. A Petri-Net based approach for verifying the integrity of production systems. *Int. J. Man-Mach. Stud.*, 36(3):447–468, 1992.

[11] Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damasio, and Gerd Wagner. Extended RDF as a Semantic Foundation of Rule Markup Languages. Technical report, TU Cottbus, October 2005.

[12] Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damasio, and Gerd Wagner. Stable model theory for extended RDF ontologies. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *Proceedings of the 4th International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science (LNCS)*, pages 21–36, Galway, Ireland, 6-10 November 2005. Springer-Verlag.

[13] Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damasio, and Gerd Wagner. On the computability and complexity issues of Extended RDF. In *Proceedings of the Tenth Pacific Rim International Conference on Artificial Intelligence (PRICAI-08)*, LNAI. Springer-Verlag, 2008.

[14] G. Antoniou. Verification and correctness issues for nonmonotonic knowledge bases. In *Proc. European Workshop on the Verification and Validation of Knowledge Based Systems*, Chambery, France, 1995.

[15] Harold Boley, Mike Dean, Benjamin Grosof, Michael Kifer, Said Tabet, and Gerd Wagner. RuleML Position Statement. In *Proceedings of W3C Workshop on Rule Languages for Interoperability*, Washington, DC, USA, 27-28 April 2005. W3C.

[16] Harold Boley, Said Tabet, and Gerd Wagner. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In *Proc. Semantic Web Working Symposium (SWWS'01)*. Stanford University, July/August 2001.

[17] Alex Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82:353–367, 1996.

[18] S. Brockmans, P. Haase, P. Hitzler, and R. Studer. A metamodel and UML profile for rule-extended OWL DL ontologies. In *In: York Sure, John Domingue (Eds.): The Semantic Web: Research and Applications, 3rd European Semantic Web Conference*, volume 4011, pages 303–316, Budva, Montenegro, 2006. Springer.

[19] Sara Brockmans. *Metamodel-based Knowledge Representation*. PhD thesis, Fakultaet fuer Wirtschaftswissenschaften der Universitaet Karlsruhe, Karlsruhe, Germany, 2007.

[20] G. Castore. Validation and verification for knowledge based control systems. In *Proceedings of the First Annual Workshop on Space Operations, Automation and Robotics, NASA,*, pages 197–202, 1987.

[21] Common Logic. Standardization project. http://suo.ieee.org/email/msg08241.html.

[22] H. Cristea, C. Kirchner, M. Moossen, and P. E. Moreau. Production systems and rete algorithm formalisation, 2004.

[23] Chris Culbert and Gary Riley. Basic programming guide, June 2003.

[24] Jos de Bruijn and Michael Kifer. F-logic/XML - An XML Syntax for F-logic, 2004.

[25] J. Dietrich and A. Paschke. On the test-driven development and validation of business rules. In *4th International Conference on Information Systems Technology and its Applications (ISTA 2005)*, New Zeeland, 2005.

[26] C. Forgy. Rete – a fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[27] Charles Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, July 1981.

[28] Ernest J. Friedman-Hill. Jess in action, 2003.

[29] N. E. Fuchs, K. Kaljurand, and G. Schneider. Attempto Controlled English Meets the Challenges of Knowledge Representation, Reasoning, Interoperability and User Interfaces. In *Proc. of FLAIRS'2006*, 2006.

[30] Rik Gerrits. Verification of business rules utilization. *Business Rules Journal*, 4(12), 2003.

[31] Adrian Giurca and Gerd Wagner. Towards an abstract syntax and direct-model theoretic semantics for RuleML. In Asaf Adi, Suzette Stoutenburg, and Said Tabet, editors, *Rules and Rule Markup Languages for the Semantic Web, First International Conference, RuleML 2005*, volume 3791 of *Lecture Notes in Computer Science*, pages 45–55, Galway, Ireland, 10-12 November 2005. Springer.

[32] T. Halpin. Business rules and object role modelling. *Database Programming and Design,*, 1996.

[33] T. Halpin. Data modeling in UML and ORM revisited, 1999.

[34] Mustafa Jarrar and Stijn Heymans. Towards pattern-based reasoning for friendly ontology debugging. *International Journal on Artificial Intelligence Tools*, 17(4), August 2008.

[35] Jess, Sandia Lab. Project Homepage. http://herzberg.ca.sandia.gov/jess/.

[36] Nima Kaviani, Dragan Gasevic, Marek Hatala, David Clement, and Gerd Wagner. Towards unifying rules and policies for semantic web services. In *Proceedings of the 3rd annual e-learning conference on Intelligent Interactive Learning Object Repositories*, Montreal, Canada, November 2006.

[37] Nima Kaviani, Dragan Gasevic, Marek Hatala, and Gerd Wagner. Web rule languages to carry policies. In *Proceedings of the 8th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 188–192, Bologna, Italy, 2007.

[38] Nima Kaviani, Dragan Gasevic, Marek Hatala, Gerd Wagner, and Timmy Eap. Exchanging policies between web service entities using rule languages. In *Proceedings of the 2007 IEEE Congress on Services (SERVICES 2007), in conjunction with, the 2007 IEEE International Conference on Web Services (ICWS 2007)*, pages 57–64, Salt Lake City, Utah, USA, 9-13 July 2007.

[39] H. W. Kelbassa and R. Knauf. The rule retranslation problem and the validation interface. In I. Russell and S. Haller, editors, *Proceedings of 16th International Florida Artificial Intelligence Research Society Conference 2003 (FLAIRS 2003)*, pages 213–217. CA: AAAI Press, 2003.

[40] H. W. Kelbassa and R. Knauf. A process approach to rule base validation and refinement. In *Proceedings of Workshop on Knowledge Engineering and Software Engineering, Koblenz, Germany, 2005*, pages 25–36. University Koblenz-Landau, Faculty of Informatics, 2005.

[41] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.

[42] Michael Kifer and James Wu. A logic for object-oriented logic programming. In *In Proc. of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 379–383, 1989.

[43] R. Knauf, A.J. Gonzalez, and T. Abel. A framework for validation of rule-based systems. *IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics*, 32:281–295, 2002.

[44] R. Knauf, S. Tsuruta, and A.J. Gonzalez. Towards reducing human involvement in validation of knowledge-based systems. In *Proceedings of the XIIIth Conference on Knowledge Acquisition and Management (KAM 2005)*, volume 1064, pages 108–112. Publishing Co. Wroclaw University of Economics, 2005.

[45] Antoni Ligeza and Grzegorz J. Nalepa. Visual design and on-line verification of tabular rule-based. In *Leipziger Informatik-Tage*, pages 303–312, 2005.

[46] N. K. Liu and T. Dillon. An approach towards the verification of expert systems using numerical Petri Nets. *International Journal of Intelligent Systems*, 6(3):255–276, 1991.

[47] John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.

[48] Sergey Lukichev. Defining a subset of OCL for expressing SWRL rules. In Adrian Giurca, editor, *Proceedings of 2nd East European Workshop on Rule-Based Applications*, CEUR Workshop Proceedings, Patras, Greece, 21-22 July 2008. CEUR.

[49] Sergey Lukichev and Adrian Giurca. Consolidation of Language Extensions, REWERSE IST 506779 Report I1-D12. Technical report, January 2008.

[50] Sergey Lukichev, Adrian Giurca, and Gerd Wagner. An integrated rule modeling framework. In *INFORMATIK 2007. Informatik trifft Logistik. Beitraege der 37. Jahrestagung der Gesellschaft fuer Informatik e.V. (GI)*, volume 109, Bremen, Germany, September 2007. GI.

[51] Sergey Lukichev and Gerd Wagner. Verbalization of the REWERSE I1 Rule Markup Language, REWERSE IST 506779 Report I1-D6. Technical report, BTU Cottbus, September 2006.

[52] Sergey Lukichev, Gerd Wagner, and Norbert E. Fuchs. Tool Improvements and Extensions 2, REWERSE IST 506779 Report I1-D11. Technical report, March 2007.

[53] D. Maier. A logic for objects. In *In Preprints of Workshop on Foundations of Deductive Databases and Logic Programming*, Washington DC, 1986.

[54] Luis Mandel and Maria Victoria Cengarle. On the expressive power of OCL. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*, pages 854–874, London, UK, 1999. Springer-Verlag.

[55] Massimo Marchiori. Metalog - towards the Semantic Web. homepage.

[56] Milan Milanovic, Dragan Gasevic, Adrian Giurca, Gerd Wagner, and Vladan Devedzic. On interchanging between OWL/SWRL and UML/OCL. In *Proceedings of the OCLApps Workshop*, pages 81–95, Genova, Italy, 2 October 2006.

[57] Milan Milanovic, Dragan Gasevic, Adrian Giurca, Gerd Wagner, and Vladan Devedzic. Model transformations to share rules between SWRL and R2ML. In *Proceedings of 3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2007)*, Innsbruck, Austria, 6 June 2007.

[58] Milan Milanovic, Dragan Gasevic, Adrian Giurca, Gerd Wagner, and Vladan Devedzic. Towards sharing rules between OWL/SWRL and UML/OCL. In *Electronic Communications of European Association of Software Science and Technology*, 2007.

[59] L. A. Miller. Dynamic testing of knowledge bases using the heuristic testing approach. *Expert Systems with Applications*, 1(3):249–269, 1990.

[60] Boris Motik, Ian Horrocks, Riccardo Rosati, and Ulrike Sattler. Can OWL and Logic Programming live together happily ever after? In *The Semantic Web - ISWC 2006*, pages 501–514. Springer Berlin / Heidelberg, 2006.

[61] Boris Motik, Ian Horrocks, and Ulrike Sattler. Adding integrity constraints to OWL. In Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors, *OWL: Experiences and Directions 2007 (OWLED 2007)*, Innsbruck, Austria, June 6–7 2007.

[62] Boris Motik, Ian Horrocks, and Ulrike Sattler. Bridging the gap between OWL and relational databases. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 807–816, New York, NY, USA, 2007. ACM.

[63] Grzegorz J. NALEPA and Antoni LIGEZA. A visual edition tool for design and verification of knowledge in rule-based systems. In *Proceedings of the 15th international conference on Systems science*, volume 3 of *Lecture Notes in Artifcial Intelligence*, pages 73–78, 2004.

[64] D. Nardi, U. Sattler, D. Calvanese, and R. Molitor. Relationships with other formalisms. In F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, editors, *Description Logic Handbook*, pages 142–183. Cambridge University Press, 2002.

[65] Oana Nicolae, Adrian Giurca, and Gerd Wagner. On interchange between JBoss Rules and Jess. In C. Badica and M. Paprzycki, editors, *Proceedings of 1st International Symposium on Intelligent and Distributed Computing*, Studies in Computational Intelligence, Craiova, Romania, 18-20 October 2007. Springer.

[66] Robert M. O'Keefe and Daniel E. O'Leary. Expert system verification and validation: a survey and tutorial. *Artificial Intelligence Review*, 7(1):3–42, 1993.

[67] Unified Modeling Language Specification. Technical Report Ver. 1.3, Object Management Group (OMG), June 1999. http://www.omg.org/uml/.

[68] OMG MOF Revision Task Force, Meta-Object Facility (MOF). Technical Report Ver. 2.0, Object Management Group (OMG), Jan 2006. http://www.omg.org/mof.

[69] OMG. Semantics of Business Vocabulary and Business Rules. Formal specification, 2008. http://www.omg.org/spec/SBVR/1.0/.

[70] Oracle views. http://oracle.com.

[71] MS OutLook, Microsoft Corp. http://www.microsoft.com.

[72] Adrian Paschke, Jens Dietrich, Adrian Giurca, Gerd Wagner, and Sergey Lukichev. On self-validating rule bases. In *Proceedings of 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006)*, Athens, GA, U.S.A, 5-9 November 2006.

[73] A. D. Preece. Foundation and application of knowledge base verification. *International Journal of Intelligent Systems*, 9(8):683–701, 1994.

[74] A. D. Preece, R. Shinghal, and A. Bakarekh. Verifying expert systems: a logical framework and a practical tool. *Exp. Stst. Appl.*, 5:421–436, 1992.

[75] RIF Basic Logic Dialect. W3C Working Draft 15 April 2008. http://www.w3.org/TR/rif-bld/.

[76] RIF Core. W3C Initial Draft, 2008. http://www.w3.org/2005/rules/wiki/Core.

[77] RIF Production Rule Dialect. W3C Working Draft 15 April 2008. http://www.w3.org/2005/rules/wiki/PRD.

[78] RIF Use Cases and Requirements. W3C Working Draft 10 July 2006. http://www.w3.org/TR/rif-ucr.

[79] Toni Rikkola. Rule analytics module, 2008. http://www.jboss.org/community/docs/DOC-11890.

[80] R. G. Ross. *Principles of the Business Rule Approach*. Addison-Wesley, 1nd edition edition, 2003.

[81] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[82] J.G. Schmolze and W. Snyder. Detecting redundancy among production rules using term rewrite semantics. *Knowledge-Based Systems*, 12:3–11, 1999.

[83] Wayne Snyder and James Schmolze. Rewrite semantics for production rule systems: Theory and applications. In Michael McRobbie and John Slaney, editors, *Proceedings 13th International Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Artifcial Intelligence*, pages 508–522. Sprinerg-Verlag, 1996.

[84] John F. Sowa. Common Logic Controlled English. Specification.

[85] Silvie Spreewenberg. Using verification and validation techniques for high-quality business rules. *Business Rules Journal*, 4(2), 2003.

[86] Standard Query Language (SQL1999). Specification 1999.

[87] Pascal Hitzler Rudi Studer and York Sure. Description logic programs: A practical choice for the modelling of ontologies. In *1st WS on Formal Ontologies meet Industry*, 2005.

[88] Said Tabet, Gerd Wagner, Silvie Spreeuwenberg, Paul D. Vincent, Gonzaques Jacques, Christian de Sainte Marie, Jon Pellant, Jim Frank, and Jacques Durand. Omg production rule representation - context and current status. In *Proceedings of W3C Workshop on Rule Languages for Interoperability*, Washington, DC, USA, 27-28 April 2005. W3C.

[89] D.J. Troy, C.T. Yu, and W. Zhang. Linearization of nonlinear recursive rules. *IEEE Transactions on Software Engineering*, 15(9):1109–1119, 1989.

[90] Gerd Wagner. Seven golden rules for a web rule language. *IEEE Intelligent Systems*, 18(5), Sep/Oct 2003.

[91] Gerd Wagner, Grigoris Antoniou, Said Tabet, and Harold Boley. The abstract syntax of RuleML - Towards a General Web Rule Language Framework. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence(WI 2004)*, pages 628–631, Beijing, China, 20-24 September 2004. IEEE Computer Society.

[92] Gerd Wagner, Adrian Giurca, and Sergey Lukichev. A usable interchange format for rich syntax rules integrating OCL, RuleML and SWRL. In *Proceedings of Reasoning on the Web*, Edinburgh, Scotland, 22 May 2006.

[93] Gerd Wagner, Adrian Giurca, Sergey Lukichev, Grigoris Antoniou, Carlos Viegas Damasio, and Norbert E. Fuchs. Language Improvements and Extensions, REWERSE IST 506779 Report I1-D8. Technical report, Munchen, Germany, April 2006.

[94] Gerd Wagner, Adrian Giurca, Sergey Lukichev, Oana Nicolae, and Mircea Diaconescu. Case Study 1: Userv product Derby 2005, REWERSE IST 506779 Report I1-D9. Technical report, TU Cottbus, March 2007.

[95] XSB. Project Homepage. http://xsb.sourceforge.net.

[96] N. Zlatareva. Truth maintenance systems and their application for verifying expert system knowledge bases. *Artificial Intelligence Review*, 6(1):67–110, 1997.

[97] N. Zlatareva. Verification of non-monotonic knowledge bases. *Eighth workshop on the validation and verification of knowledge-based systems*, 21(4):253–261, 1997.